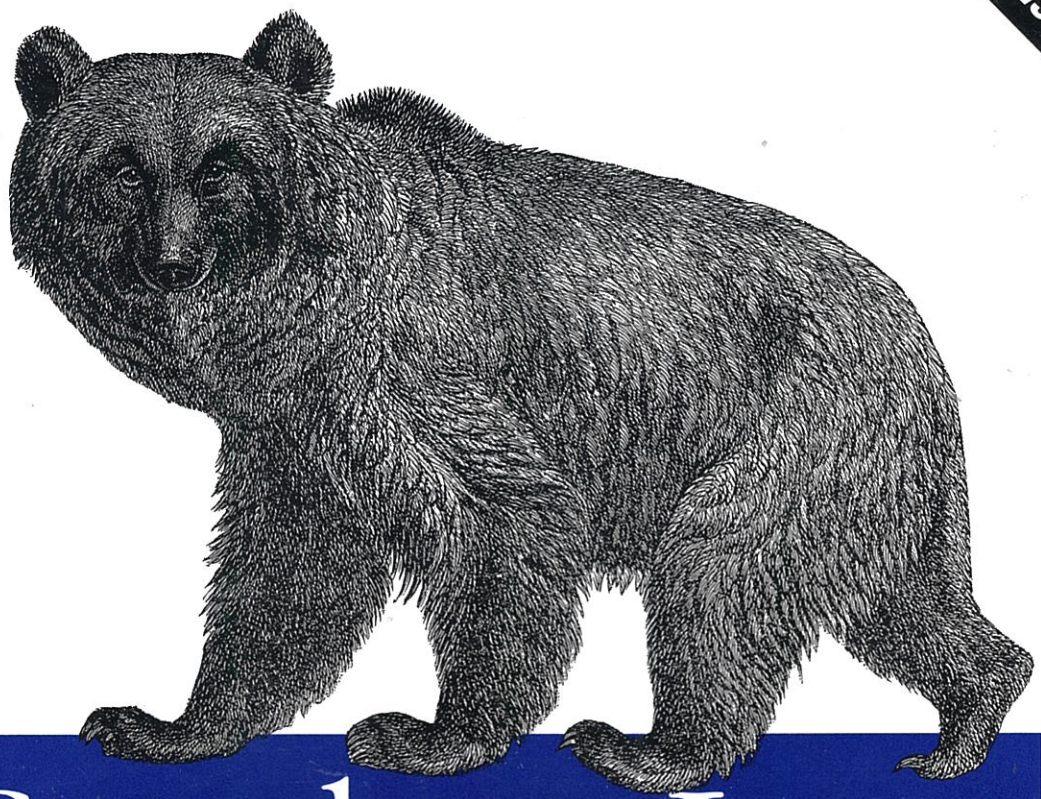
2e édition en français



Servlets Java™

Guide du programmeur

Jason Hunter avec William Crawford
Traduction de Hervé Soulard

O'REILLY®

Servlets Java

2e édition

JASON HUNTER

avec WILLIAM CRAWFORD

Servlets Java

Guide du programmeur

2e édition

Traduction de HERVÉ SOULARD

Éditions O'REILLY
18 rue Séguier
75006 Paris
france@ora.com
http://www.editions-oreilly.fr/

O'REILLY™

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

L'édition originale de ce livre a été publiée aux États-Unis par O'REILLY & Associates sous le titre *Java Servlet Programming, 2nd Edition*, ISBN 0-596-00040-5

© O'REILLY & Associates, 2001

Historique :
- Première édition : octobre 1998
- Deuxième édition : avril 2001

Couverture conçue par Hanna Dyer

Édition : Dominique BURAUD

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

© ÉDITIONS O'REILLY, Paris, 2002
ISBN 2-84177-196-2

Historique :
- Première édition : juillet 1999
- Deuxième édition : décembre 2002

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

Table des matières

Préface ..... *ix*

1. Introduction *1*
Histoire des applications web 1
Support des servlets 6
Puissance des servlets 10

2. Introduction aux servlets HTTP *13*
Bases de HTTP 13
L'API Servlet 16
Générer des pages 18
Applications web 24
Allez plus loin 31

3. Cycle de vie d'une servlet *33*
La solution servlet 33
Rechargement d'une servlet 39
Init() et destroy() 40
Modèle mono-thread 48
Traitement en arrière-plan 50
Chargement au démarrage 52
Mise en cache côté client 54
Mise en cache côté serveur 56

4. Retrouver l'information *69*
La servlet 71
Le serveur 73

Le client 82
La requête 87

5. Envoyer une information HTML 123
Structure d'une réponse 123
Envoyer une réponse normale 124
Utiliser des connexions persistantes 126
Mettre la réponse dans un tampon 127
Codes d'état 129
En-têtes HTTP 132
Lorsque tout va mal 139
Six façons d'habiller une servlet 151

6. Envoyer un contenu multimédia 155
WAP et WML 155
Images 165
Contenu compressé 183
Push du serveur 186

7. Suivi de session 191
Authentifier l'utilisateur 192
Champs de formulaire cachés 193
Réécrire une URL 196
Cookies persistants 198
L'API de suivi de session 203

8. Sécurité 219
Authentification HTTP 220
Authentification à base de formulaire 227
Authentification personnalisée 230
Certificats numériques 237
Secure Sockets Layer (SSL) 239

9. Se connecter aux bases de données 247
Bases de données relationnelles 249
L'API JDBC 251
Réutiliser des objets de base de données 265
Transactions 267
Une servlet Livre d'or 277
Techniques JDBC avancées 282
API JDBC avancée 285

10. Communication applet-servlet 287
Options de communication 287
Serveur horaire 293
Serveur de discussion 327

11. Collaboration de servlets 347
Partage d'informations 347
Partage du contrôle 351

12. Servlets d'entreprise et J2EE 361
Distribuer la charge 361
Intégration avec J2EE 365

13. Internationalisation 371
Langues de l'Europe de l'ouest 372
Se conformer aux habitudes locales 375
Langues hors Europe de l'ouest 377
Langues multiples 381
Négociation dynamique de la langue 384
Formulaires HTML 394

14. L'environnement Tea 399
Le langage Tea 400
Démarrage 401
Informations de requête 403
Administration de Tea 406
Applications Tea 410
Une application outil 416
Pour finir 428

15. WebMacro 429
L'environnement WebMacro 430
Installer WebMacro 433
Directives de WebMacro 439
Modèles WebMacro 442
Une application outil 447
Filtres 452

16. Element Construction Set 455
Composants d'une page sous forme d'objets 455
Afficher un ensemble de résultats 457

17. XMLC 469
Une compilation XML simple 470
La classe de manipulation 476
Une application outil 480

18. JavaServer Pages 489
Utiliser les JavaServer Pages 490
En coulisses 492
Expressions et déclarations 494
Directives 495
JSP et JavaBeans 500
Inclusions et renvois 504
Une application outil 506
Bibliothèques de balises personnalisées 511

19. Affaires diverses 517
Analyser les paramètres 517
Envoyer un e-mail 522
Utiliser les expressions régulières 525
Exécuter des programmes 529
Utiliser les méthodes natives 534
Agir comme un client RMI 534
Débogage 536
Ajuster les performances 544

20. Quoi de neuf dans l'API Servlet 2.3 ? 547
Changements dans l'API Servlet 2.3 547
Conclusion 561

A. L'API Servlet, référence rapide 563

B. L'API Servlet HTTP, référence rapide 593

C. Descripteur de déploiement, référence de la DTD 621

D. Codes d'état HTTP 651

E. Entités caractères 659

F. Charsets 665

Index 667

Préface

Depuis l'écriture de la première édition de ce livre, l'adoption des servlets et de la plate-forme Java côté serveur a été bien plus importante que ne le prévoyaient les prévisions les plus optimistes. Les fournisseurs de serveurs web offrent aujourd'hui en standard le support des servlets. La spécification de Java 2, Enterprise Edition (J2EE), a fait des servlets un composant interne et les fournisseurs de serveurs d'applications ne seraient pas pérennes sans une implémentation évolutive des servlets. Mais c'est plus qu'un matraquage publicitaire des fournisseurs. Les servlets sont devenues la base des JavaServer Pages (JSP) et d'autres frameworks, et la technologie des servlets est utilisée aujourd'hui sur des sites à fort trafic tels que ESPN.com et AltaVista.com.

Il n'est pas surprenant que le monde des servlets ressemble peu aujourd'hui à ce qu'il était au moment de l'impression de la première édition. L'API Servlet a subi deux révisions, la troisième est en cours. De jeunes sociétés comme Live Software et New Atlanta qui ont gagné de l'argent en vendant les moteurs de servlets (appelés maintenant des *conteneurs de servlets*) JRun et ServletExec se sont fait remarquer et ont été rachetées par des entreprises web plus importantes, respectivement Allaire et Unify. Elles offrent maintenant des fonctionnalités qui vont plus loin que le support de base des servlets afin de se différencier.

De manière assez surprenante, les paquetages officiels `javax.servlet` et `javax.servlet.http` ont été les premières classes Java à être officiellement disponibles en *open source*. Ils ont été donnés à l'ASF (*Apache Software Foundation*) et ils se trouvent maintenant sur le site *http://jakarta.apache.org*. Les paquetages continuent de respecter la spécification de l'API Servlet, mais les corrections d'erreurs et les mises à jour de la spécification peuvent maintenant se faire par une élite de développeurs — dont votre humble serviteur qui a récemment réussi à améliorer la gestion des requêtes GET dans `HttpServlet`. De plus, le serveur qui sert d'implémentation de référence de l'API Servlet a aussi été donné à l'ASF et rendu disponible en *open source* sous le nom Apache Tomcat. Tomcat est devenu depuis l'un des conteneurs de servlets les plus répandus. Pour plus d'informations, allez sur le site *http://opensource.org*.

Le monde des servlets a changé et le but de ce livre est de vous maintenir à jour. Il explique tout ce que vous devez savoir sur la programmation des servlets Java, du début à

la fin. Les cinq premiers chapitres concernent les bases : ce que sont les servlets, ce qu'elles font et comment elles fonctionnent. Les quinze chapitres suivants contiennent la vraie matière : ils explorent ce que vous ferez certainement avec les servlets et les outils que vous utiliserez probablement. Vous trouverez de nombreux exemples, plusieurs suggestions, quelques avertissements et même un ou deux véritables *hacks* qui, d'une certaine façon, ont permis à ce livre de passer les relectures techniques.

L'API Servlet 2.2

Cette édition concerne la version 2.2 de l'API Servlet qui a été rendue publique en août 1999 et dont la version finale est sortie en décembre 1999. La première édition de ce livre concernait la version 2.0. Les modifications survenues entre les deux versions sont importantes :

- Des règles de distribution des servlets sur de multiples serveurs ont été édictées.
- Les servlets peuvent maintenant exploiter les applications web déployables, que l'on peut configurer et déployer de manière indépendante du serveur.
- La sécurité des servlets a été grandement améliorée.
- Les servlets peuvent maintenant déléguer la gestion des requêtes à d'autres composants serveur.
- Les servlets peuvent maintenant partager des informations *via* leur `ServletContext`.
- Il existe un moyen de rendre abstraites les ressources représentées par les servlets afin de supporter les accès distribués.
- Les servlets ont maintenant un contrôle plus important sur la gestion des sessions.
- La mise en tampon de la réponse a été ajoutée.
- Le contrôle des en-têtes HTTP a été étendu.
- Il est possible d'employer une gestion des erreurs plus avancée.
- L'API a été « nettoyée » afin que les noms de méthodes soient plus cohérents et prévisibles.
- L'API Servlet est maintenant définie dans un document de spécification formelle et les mises à jour futures de l'API se font par un processus JSR (*Java Specification Request*) formel.
- Les servlets se trouvent maintenant au cœur de la spécification de la plate-forme J2EE (*Java 2 Enterprise Edition*).

Toutes ces modifications, ainsi que d'autres changements mineurs, sont totalement expliquées dans cette nouvelle édition. Elle couvre aussi un aspect très intéressant du développement de servlets : des techniques de construction reposant sur les servlets pour créer facilement et efficacement du contenu dynamique dans des sites grandeur nature. Dans cette édition, vous trouverez des tutoriels sur les cinq technologies de création de contenu basées sur les servlets les plus répandues :

- les JSP (*JavaServer Pages*), un standard de Sun, développées et sorties en même temp que les servlets ;

- Tea, créé par Walt Disney Internet Group (précédemment GO.com) et utilisé par des sites à fort trafic comme ESPN.com, NFL.com, Disney.com, DisneyLand.com, GO.com et Movies.com ;
- WebMacro, créé par Semiotek et utilisé par le moteur de recherche AltaVista ;
- XMLC, créé par Lutris Technologies afin d'exploiter la puissance de la technologie XML sur le web et utilisé par des sites innovants comme customatix.com ;
- ECS (*Element Construction Set*), créé par Apache pour gérer les besoins de programmation les plus souvent recontrés.

Cette seconde édition concerne aussi WAP (*Wireless Application Protocol*) et elle explique comment créer des applications web basées sur les servlets et destinées aux périphériques sans fil.

L'API Servlet 2.3

Au moment de l'écriture de ce livre, l'API Servlet 2.3 est en cours de développement. Elle n'est cependant pas finalisée. Par conséquent, le contenu de cette édition inclut à différents endroits de brèves remarques sur les modifications attendues dans l'API Servlet 2.3. De plus, le dernier chapitre de ce livre décrit en détail la spécification dans l'état où elle était en octobre 2000 afin que vous puissiez vous familiariser avec les apports de l'API Servlet 2.3. Nous devez cependant noter que ces spécifications risquent d'évoluer et que la version finale pourrait être légèrement différente de ce que l'on présente ici.[1]

Aux lecteurs de la première édition

Les lecteurs de *Servlets Java, Guide du programmeur, 1re édition* trouveront que ce livre a été totalement mis à jour pour l'API Servlet 2.2 et, quand c'est possible, pour l'API Servlet 2.3. Chaque chapitre a subi des améliorations importantes. Six nouveaux chapitres concernent les techniques de création de contenu à base de servlets et un septième, *Servlets d'entreprise et J2EE,* explique comment les servlets s'intègrent à la plate-forme J2EE.

Puisque le modèle d'*application web* déployable a un impact transversal sur tous les aspects de la programmation des servlets, nous recommandons aux lecteurs de la première édition de lire chaque chapitre et de prendre note des nouveaux mécanismes permettant d'accomplir les tâches traditionnelles. Pour les lecteurs dont le temps est limité, nous avons ajouté dans la section *Organisation* une liste des modifications les plus importantes.

Notre audience

Ce livre est-il pour vous ? Il l'est si vous souhaitez développer des applications à déployer sur le Web. Plus particulièrement, ce livre a été écrit pour aider :

---

1. *N.d.T.* : Au moment de la traduction de ce livre, la version finale de l'API Servlet 2.3 est disponible. Cependant, d'après les auteurs, les changements apportés et l'organisation du livre ne remettent pas en cause le contenu de cette seconde édition.

Les programmeurs J2EE

Les servlets font intégralement partie de la norme Java 2, Enterprise Edition. Les programmeurs développant pour des serveurs J2EE peuvent apprendre comment intégrer au mieux les servlets dans d'autres technologies *backend*.

Les programmeurs JSP

Les JSP (*JavaServer Pages*) sont construites au-dessus des servlets. Pour pleinement exploiter les JSP, il faut comprendre les servlets, ce qu'apporte ce livre. Il inclut aussi un tutoriel sur les JSP ainsi que sur les quatre principales alternatives aux JSP.

Les programmeurs d'applets Java

Il a toujours été difficile pour une applet de communiquer avec un serveur. Les servlets facilitent cette communication en offrant à une applet un agent écrit en Java et mis sur le serveur, auquel elle peut facilement se connecter.

Les programmeurs CGI

CGI est un moyen bien connu d'étendre les fonctionnalités d'un serveur Web. Les servlets offrent une autre possibilité élégante et efficace.

Les autres programmeurs côté serveur

Il existe plusieurs alternatives à CGI, dont FastCGI, PHP, NSAPI, WAI, ISAPI, ASP et maintenant ASP+. Chacune a des limitations en ce qui concerne la portabilité, la sécurité et/ou les performances avec les sources de données *backend*. Les servlets ont tendance à exceller dans chacun de ces domaines.

Prérequis

Quand nous avons commencé à écrire ce livre, à notre grande surprise, nous avons découvert que l'une des choses les plus difficiles était de déterminer les connaissances supposées du lecteur. Êtes-vous familier de Java ? Avez-vous déjà programmé en CGI ou réalisé des applications pour le Web ? Est-ce la première fois que vous mettez les pieds dans le monde des servlets ? Comprenez-vous HTTP et HTML, ou ces acronymes vous semblent-ils interchangeables ? Qu'importe le niveau d'expérience que nous pouvions imaginer, il aurait de toute façon été trop élevé pour certains ou trop faible pour d'autres.

Finalement nous avons écrit ce livre en souhaitant y mettre au maximum un contenu original : on peut omettre une description exhaustive des sujets et concepts qui sont bien décrits en ligne ou dans d'autres livres. Tout au long du texte, vous trouverez plusieurs références à des sources d'information externes.

Bien entendu, des références externes vous éloignent. Ce livre suppose que vous connaissez plutôt bien le langage Java et les bases des techniques de programmation orientées objet. Si vous arrivez aux servlets à partir d'un autre langage, nous vous suggérons de commencer par lire un livre général sur la programmation Java, comme *Introduction à Java* de Patrick Niemeyer et Jonathan Knudsen (Éditions O'Reilly). Vous pourrez passer rapidement sur les sections concernant les applets et Swing pour passer plus de temps sur la programmation réseau et le multithread. Si vous voulez démarrez immédiatement avec les servlets et apprendre Java au fur et à mesure que vous avancez, nous vous suggérons de lire ce livre en gardant près de vous une copie de *Java in a Nutshell* de David Flanagan (Éditions O'Reilly), ou tout autre livre de référence sur Java.

Ce livre ne suppose pas que vous avez une grande expérience de la programmation web, de HTTP et de HTML. Mais il ne fournit pas non plus une introduction complète ni une description exhaustive de ces technologies. Nous en verrons les bases nécessaires au développement de servlets mais nous n'aborderons pas les points de détail (tels que la liste complète des balises HTML et les en-têtes HTTP 1.1).

À propos des exemples

Vous trouverez dans ce livre plus d'une centaine d'exemples de servlets. Tout le code est inclus dans le texte, mais vous pourriez préférer télécharger les exemples plutôt que de les entrer à la main. Vous trouverez le code en ligne et prêt au téléchargement à *http://www.oreilly.com/catalog/jservlet2/*. Vous pouvez aussi voir plusieurs des servlets en action à *http://www.servlets.com*.

Tous les exemples ont été testés avec le serveur Apache Tomcat 3.2 s'exécutant sur la machine virtuelle Java (JVM) livrée avec les versions 1.1.8 et 1.2.2 du Java Development Kit (JDK), sous Windows et Unix. Quelques exemples avancés demandent des fonctionnalités que Tomcat ne supporte pas en mode autonome ; dans ce cas, les exemples ont été testés avec d'autres serveurs, comme il est indiqué dans le texte. Apache Tomcat est l'implémentation de référence de l'API Servlet. Il est disponible gratuitement sous forme de licence en *open source* sur le site *http://jakarta.apache.org*.

Ce livre contient aussi un ensemble de classes utilitaires — elles sont utilisées pour les exemples de servlets et vous pourriez les trouver utiles pour votre propre développement. Ces classes sont contenues dans le package `com.oreilly.servlet`. Entre autres choses, il y a des classes qui aident les servlets à analyser les paramètres, gérer les téléchargements de fichiers, générer des réponses multi-parties (*push* de serveur), négocier des *locales* pour l'internationalisation, retourner des fichiers, gérer des connexions par socket, et agir en tant que serveur RMI. Il y a même une classe qui facilite la communication entre les applets et les servlets. Depuis la première édition de ce livre, de nouvelles classes ont été ajoutées afin d'aider les servlets à envoyer des courriers électroniques, à mettre la réponse dans un cache et à détecter automatiquement le support de l'API des servlets. Le code source du package `com.oreilly.servlet` est contenu dans le texte et la dernière version complète est disponible (avec la documentation *javadoc*) en ligne sur le site *http://www.servlets.com*.[2]

2. Ce livre ne vient pas avec un CD-Rom. L'ajout d'un CD-Rom augmente le coût de production et vous coûte de l'argent. Nous pensons que toute personne lisant ce livre dispose d'un accès à Internet et que vous économiserez de l'argent en téléchargeant le code des exemples sur le Web. De plus, nous ne voyons aucun intérêt à inclure des versions d'évaluation des différents serveurs web et d'applications. Si l'on considère l'allure folle à laquelle le marché des servlets innove, les serveurs inclus sur un CD-Rom seraient obsolètes avant que le livre n'arrive chez l'imprimeur. Les mêmes versions d'évaluation sont disponibles en ligne et nous vous recommandons de télécharger les dernières versions. Souvenez-vous simplement que si vous voulez lire le livre hors-ligne, vous devrez télécharger le code des exemples et une copie du serveur web Apache Tomcat. Les liens pour le téléchargement se trouvent sur le site *http://www.servlets.com*.

Organisation

Ce livre est constitué de 20 chapitres et de 6 annexes :

Chapitre 1, *Introduction*

Explique le rôle et les avantages des servlets Java dans le développement d'applications web. La seconde édition a été complétée par d'autres informations sur les serveurs.

Chapitre 2, *Introduction aux servlets HTTP*

Fournit une introduction rapide à HTTP et aux choses que les servlets HTPP peuvent faire. Il montre la génération de page web basique et introduit la notion d'application web déployable. La seconde édition couvre les applications web et leurs descripteurs de déploiement XML.

Chapitre 3, *Cycle de vie d'une servlet*

Explique en détail comment et quand une servlet est chargée, comment et quand elle est exécutée, comment les threads sont gérées et comment traiter les problèmes de synchronisation dans un système multithread. La gestion de la persistance des états est aussi expliquée. La seconde édition inclut les nouvelles règles pour le rechargement de contexte et l'enregistrement des servlets, une nouvelle section sur la mise en cache côté serveur et un nouvel encadré sur `super.init(config)`.

Chapitre 4, *Retrouver l'information*

Introduit les méthodes les plus classiques qu'une servlet utilise pour recevoir de l'information — depuis le client, le serveur, la requête du client, ou elle-même. Il montre aussi une classe réutilisable pour le déchargement de fichiers (*upload*). La seconde édition explique comment placer les informations dans le descripteur de déploiement, comment obtenir le nom d'une servlet, comment accéder aux répertoires temporaires, comment gérer les paramètres d'initialisation du contexte, comment déterminer la version de l'API Servlet, comment définir des correspondances de servlet et comment accéder aux ressources abstraites. Elle montre aussi un composant de déchargement de fichiers amélioré et plus souple.

Chapitre 5, *Envoyer une information HTML*

Décrit comment une servlet peut générer du HTML, retourner des erreurs et d'autres codes d'état, rediriger des requêtes, écrire des données dans le journal du serveur, et envoyer des en-têtes HTTP personnalisés. La seconde édition comporte une nouvelle description de la mise en tampon de la réponse, un exemple de redirection vraiment utile et de nouvelles section sur la configuration des pages d'erreurs et sur la gestion des erreurs.

Chapitre 6, *Envoyer un contenu multimédia*

Examine certaines des choses intéressantes qu'une servlet peut retourner : du contenu WAP/WML pour les périphériques sans fil, des images générées dynamiquement, du contenu compressé et des réponses multi-parties. La seconde édition ajoute la description de WAP/WML, les listes de fichiers de bienvenue, une présentation de PNG, un cache d'images côté serveur amélioré et plus de détails sur la compression du contenu.

Chapitre 7, *Suivi de session*

Montre comment construire une forme d'état au-dessus du protocole sans état HTTP. La première moitié du chapitre montre les techniques habituelles de suivi de session utilisées par les développeurs CGI ; la seconde moitié montre comment utiliser le support intégré de suivi de session de l'API Servlet. La seconde édition inclut des règles pour les sessions des applications web, du contenu sur les noms des nouvelles méthodes de suivi de session, une présentation sur la gestion des échéances (*timeout*) et le suivi de session basé sur les applets.

Chapitre 8, *Sécurité*

Explique les problèmes de sécurité liés à l'informatique distribuée. Il montre comment utiliser les servlets pour la gestion des comptes utilisateur et comment construire un système plus performant basé sur l'authentification et l'autorisation. Il explique aussi le rôle des servlets dans les communications SSL sécurisées. Il a été complètement mis à jour dans la seconde édition.

Chapitre 9, *Se connecter aux bases de données*

Montre comment les servlets peuvent être utilisées pour des connexions web/base de données à haute performance. Il inclut un tutorial sur JDBC. La seconde édition contient une démonstration de la configuration des connexions à partir d'un fichier de propriétés, un nouvel exemple du livre d'or et une nouvelle section sur JDBC 2.0.

Chapitre 10, *Communication applet-servlet*

Décrit comment les servlets peuvent être utiles aux applets qui ont besoin de communiquer avec le serveur. Il a été mis à jour dans cette seconde édition.

Chapitre 11, *Collaboration de servlets*

Explique pourquoi les servlets ont besoin de communiquer les unes avec les autres et comment cela peut se faire par un partage d'informations ou par des appels inter-servlets. Il a été complètement revu dans cette seconde édition.

Chapitre 12, *Servlets d'entreprise et J2EE*

Présente les fonctionnalités avancées des servlets destinés aux sites d'entreprise : distribution de la charge et intégration des composants J2EE. Nouveau dans cette seconde édition.

Chapitre 13, *Internationalisation*

Montre comment une servlet peut générer du contenu multilingue. La seconde édition explique comment utiliser *javac* pour gérer les codages et comment utiliser les nouvelles méthodes de l'API pour gérer les locales.

Chapitre 14, *L'environnement Tea*

Présente le framework Tea, un moteur de modèle puissant et élégant. Nouveau dans cette seconde édition.

Chapitre 15, *WebMacro*

Présente le framework WebMacro, similaire à Tea mais avec plusieurs choix de conception différents. Nouveau dans cette seconde édition.

Chapitre 16, *Element Construction Set*

Décrit brièvement ECS, une approche orientée objet à la création de pages. Nouveau dans cette seconde édition.

Chapitre 17, *XMLC*

Donne une vue d'ensemble de XMLC, une approche à la création de pages basée sur XML. Nouveau dans cette seconde édition.

Chapitre 18, *JavaServer Pages*

Décrit les JSP, la technologie standard de Sun dans laquelle les pages web sont autocompilées en servlets. Nouveau dans cette seconde édition.

Chapitre 19, *Affaires diverses*

Présente un bric à brac d'exemples de servlets utiles et des astuces qui n'ont pas trouvé de place ailleurs. La seconde édition inclut un analyseur de paramètres localisés, une nouvelle classe pour le courrier électronique, une section sur les expressions régulières mise à jour, une nouvelle section sur les outils tiers et d'autres astuces pour l'optimisation des performances.

Chapitre 20, *Quoi de neuf dans l'API Servlet 2.3 ?*

Illustre les modifications à venir dans la version 2.3 de l'API Servlet prévue pour la mi-2001. Nouveau dans cette seconde édition.

Annexe A, *L'API Servlet, référence rapide*

Contient une description complète des classes, méthodes et variables du paquetage `javax.servlet`. La seconde édition a été mise à jour pour l'API Servlet 2.2.

Annexe B, *L'API Servlet HTTP, référence rapide*

Contient une description complète des classes, méthodes et variables du paquetage `javax.servlet.http`. La seconde édition a été mise à jour pour l'API Servlet 2.2.

Annexe C, *Descripteur de déploiement, référence de la DTD*

Présente une référence annontée de la DTD (*Document Type Definition*) du descripteur de déploiement *web.xml*. Nouveau dans cette seconde édition.

Annexe D, *Codes d'état HTTP*

Liste les codes d'état spécifiés par HTTP, avec les constantes mnémoniques utilisées par les servlets.

Annexe E, *Entités caractères*

Liste les entités caractères définies en HTML, avec leur équivalent Unicode.

Annexe F, *Charsets*

Liste les tables de caractères que les servlets peuvent utiliser pour générer du contenu en plusieurs langues.

N'hésitez pas à lire les chapitres de ce livre dans l'ordre que vous voulez. Les lire du premier au dernier garantit que vous n'aurez aucune mauvaise surprise puisque nous avons fait des efforts pour éviter les références en avant. Cependant, si vous souhaitez gambader d'un chapitre à l'autre, c'est possible, surtout après le chapitre 5 — les autres chapitres étant plutôt autonomes. Dernière suggestion : lisez la section *Débogage* du chapitre 19 si, à quelque moment que ce soit, vous tombez sur un morceau de code qui ne marche pas comme vous l'espériez.

Conventions

Le texte en *italique* est utilisé pour :

- Noms de chemins, noms de fichiers et noms de programmes.
- Nouveaux termes au moment de leur définition.
- Adresses Internet, comme des noms de domaine ou des URL.

Le texte en gras est utilisé pour :

- Des touches particulières sur un clavier d'ordinateur.
- Les noms de boutons ou de menus de l'interface utilisateur.

Le texte en `chasse constante` est utilisé pour :

- Tout ce qui apparaît littéralement dans un programme Java, y compris les mots-clés, les types de données, les noms de méthode, les variables, les noms de classe et les noms d'interface.
- Tous les listings de code Java.
- Les documents, les balises et les attributs HTML.

Le texte en *`chasse constante italique`* est utilisé pour :

- Les emplacements généraux qui indiquent que quelque chose doit être remplacé par une valeur réelle dans votre programme.

Le texte en **`chasse constante grasse`** est utilisé pour :

- Les entrées de la ligne de commande.

Votre avis

Aidez-nous à améliorer les futures éditions de ce livre en nous indiquant les erreurs, les bogues, les affirmations induisant en erreur ou confuses et les fautes d'orthographe que vous pourriez trouver dans ce livre. Envoyez vos rapports de bogues et vos commentaires par email à *france@oreilly.com*. (Cependant, avant d'envoyer votre rapport, vous devriez jeter un œil à la liste d'errata à *http://www.oreilly.com/catalog/jservlet2* pour voir si le bogue n'a pas déjà été signalé.)

Indiquez-nous aussi ce qui pourrait rendre ce livre plus utile à vos yeux. Vos commentaires nous sont réellement précieux et nous essaierons d'intégrer les suggestions raisonnables dans les futures éditions.

Remerciements

Pendant que je travaillais sur cette seconde édition, un ami m'a dit : « Il doit t'être plus facile d'écrire une seconde édition ; tu as déjà écrit le livre une fois ». J'ai réfléchi une minute, ai rigolé et répondu : « C'est plus facile, mais pas forcément aussi facile que je l'espérai ! »

En y repensant, je crois que la raison n'est pas liée aux livres mais à la technologie. La première édition concernait l'API Servlet 2.0, une spécification développée il y a trois

ans. Cette seconde édition concerne les API Servlet 2.2 et 2.3 qui représentent près de trois ans de travail de conception. On comprend ainsi facilement que si la première édition a demandé un an d'écriture intensive, alors la seconde édition doit prendre quasiment le même temps. Et cela a été le cas : près de neuf mois.

Je suis profondément reconnaissant à toutes les personnes qui m'ont aidé à écrire ce livre. Tout d'abord, ses relecteurs techniques : James Duncan Davidson, acteur principal de la spécification de l'API Servlet pour les version 2.1 et 2.2, et Danny Coward, acteur principal de l'API 2.3. Je ne remercierai jamais assez James et Danny. Non seulement ils m'ont fourni une assistance et des conseils inestimables tout au long de l'écriture de ce livre, mais ils nous ont donné une plate-forme merveilleuse pour la programmation web.

Un grand merci à tous les développeurs qui ont apporté leur expertise dans les chapitres concernant la création du contenu (et dans beaucoup de cas qui ont créé la technologie présentée) : Reece Wilton et Brian O'Neill pour Tea, Justin Wells pour WebMacro, Jon Stevens pour ECS, Mark Diekhans et Christian Cryder pour XMLC et Hans Bergsten et Craig McClanahan pour les JSP.

Je veux aussi remercier Bob Eckstein, l'éditeur de ce livre, dont les remarques manuscrites étaient toujours perspicaces et parfois illisibles. Bob a pris en charge l'édition après que Paula Ferguson soit devenue responsable de la gestion du site web d'O'Reilly et des livres sur les scripts.

Merci aussi à Jim Grisham, qui m'a aidé à trouver différents types de machines et de navigateurs pour tester les exemples ; à Magnus Stenman d'Orion, qui m'a expliqué comme Orion Server implémente J2EE ; à Justyna Horwat, appelée par certains la déesse de la bibliothèque de balises, pour avoir répondu aux questions sur les bibliothèques de balises JSP ; et à Ethan Henry, pour ses suggestions sur l'ajustement des performances des servlets.

Je ne peux oublier Brett McLaughlin, auteur de *Java et XML* (Éditions O'Reilly) et co-créateur de JDOM. Notre coopération sur JDOM a en fait ralenti l'écriture du livre mais la vitesse à laquelle il écrit m'a inspiré et, avec les remerciements qu'il m'a donné dans son livre, je dois lui dire *quelque chose* ici !

Et enfin, merci à mon amie, Kathlyn Bautista, qui ne s'est pas plainte quand je travaillais les dimanches mais qui ne m'y a jamais poussé.

Jason Hunter
Novembre 2000

Remerciements de la première édition

En un sens, ce livre a commencé le 20 mars 1997, à la librairie Computer Literacy à San Jose, en Californie. Là — après une crise de rires avec Larry Wall et Randall Schwartz, où Larry a expliqué comment il envisage d'automatiser sa maison avec Perl — j'ai rencontré pour la première fois le très estimé Tim O'Reilly. Je me suis présenté et je lui ai dit très effrontément qu'un jour (très loin dans le futur, ai-je pensé) j'écrirais un livre pour O'Reilly. Je me sentais comme si je disais à Steven Spielberg que j'envisageais de jouer dans l'un de ses films. À ma très grande surprise, Tim a répondu « Sur quel sujet ? ». C'est ainsi qu'a commencé le parcours du combattant conduisant à ce livre.

Il y a plusieurs points importants dont je me souviens : ma rencontre avec mon éditeur (cool, elle est jeune, elle aussi !), la signature du contrat officiel (savez-vous que les papiers officiels de O'Reilly ont des animaux dessus ?), l'écriture de la première phrase (encore et encore), l'impression du premier chapitre (et faire en sorte qu'il ressemble à un livre O'Reilly), et ensuite regarder les sorties papier s'empiler de plus en plus haut, jusqu'à ce qu'il n'y ait plus rien à écrire (sauf les remerciements).

Il y a eu aussi beaucoup d'essais. À un moment donné, lorsque le livre était à moitié fini, j'ai réalisé que l'API Servlet changeait plus vite que je ne pouvais suivre. Je crois en la maxime « Si vous ne réussissez pas du premier coup, demandez de l'aide ». Après une rapide recherche de talents, j'ai demandé à William Crawford, qui avait déjà travaillé sur *Java Enterprise in a Nutshell*, s'il pouvait m'aider à accélérer la fin de la rédaction du livre. Il a généreusement accepté et il a fini par écrire deux chapitres, ainsi que des portions des annexes.

Il y a plusieurs autres personnes qui ont aidé à l'écriture de ce livre, directement ou indirectement. Je voudrais remercier Paula Ferguson, l'éditeur du livre, et Mike Loukides, l'éditeur de la *Java series*, pour les efforts qu'ils ont faits afin de garantir (et améliorer) la qualité de ce livre. Et à Tim O'Reilly pour m'avoir donné la chance de réaliser un rêve.

Merci aussi à mes directeurs chez Silicon Graphics, Kathy Tansill et Walt Johnson, pour m'avoir donné plus d'encouragements et de souplesse que j'étais en droit d'espérer.

Je ne remercierai jamais assez les ingénieurs de Sun qui m'ont vraiment aidé en répondant à mes questions, en me gardant au courant des changements sur l'API Servlet, et en corrigeant rapidement à peu près tous les bogues que je leur ai soumis : James Duncan Davidson (qui ressemble à la marionnette de James Gosling), Jim Driscoll, Rob Clark et Dave Brownell.

Merci aussi aux membres de la liste de diffusion *jserv-interest*, dont les questions et les réponses ont formé le contenu de ce livre ; Will Ramey, un vieil ami qui n'a pas laissé l'amitié aveugler son œil critique ; Mike Engber, l'homme vers qui je me retournais lorsque j'étais à court de détours élégants et étais prêt à accepter les choses folles qu'il pouvait dire ; Dave Vandegrift, la première personne à lire les chapitres ; Bill Day, auteur de *Java Media Players*, qui m'a aidé en suivant le processus d'écriture du livre en parallèle avec moi ; Michael O'Connell et Jill Steinberg, éditeurs pour *JavaWorld*, où j'ai fait mes premiers écrits professionnels ; Doug Young, qui a partagé avec moi les astuces qu'il a apprises en écrivant seul sept livres techniques ; et Shoji Kuwabara, Mieko Aono, Song Yung, Matthew Kim et Alexandr Pashintsev pour leur aide sur la traduction de « Bonjour tout le monde » dans le chapitre 12.

Je voudrais aussi remercier les relecteurs techniques de ce livre, dont les critiques constructives ont fait plus qu'améliorer ce travail : Mike Slinn, Mike Hogarth, James Duncan Davidson, Dan Pritchett, Dave McMurdie et Rob Clark. Je suis encore étonné qu'il n'ait pris que trois jours à un relecteur pour lire ce qui a pris un an d'écriture !

Et enfin, merci à Maman et Papa, pour leur amour et leur soutien et pour le temps qu'ils ont passé il y a longtemps à m'apprendre les bases de l'écriture. Et un remerciement tout spécial à Kristi Taylor qui a transformé en plaisir le peu de temps en dehors du travail.

Et Grand-Papa, j'aurai voulu que tu voies cela.

Jason Hunter
Juin 1998

Tout d'abord et en premier, merci à Shelley Norton, Dr. Isaac Kohane, Dr. James Fackler et Dr. Richard Kitz (plus un club de supporters dont les contributions ont été inestimables), dont l'assistance et le soutien précoce ont rendu les choses possibles. Et aussi à Martin Streeter de Invantage, Inc., pour son soutien tout au long de ce projet.

Sans Rob Leith, Roger Stacey et Fred Strebeigh, toutes mes phrases seraient encore probablement à la voix passive. Dale Dougherty m'a offert de l'argent en échange de mots, ce dont je ne reviens toujours pas. Andy Kwak, Joel Pomerantz et Matthew Proto, tous de valeureuses personnes, qui ont consenti à lire les versions préliminaires et à écouter mes lamentations à une heure du matin.

Et bien sûr, à maman et à papa pour leurs années de soutien, et à ma sœur Faith pour ne pas (trop) m'en vouloir d'être un nerd.

William Crawford
Juillet 1998

1

Introduction

L'arrivée des applications Java sur le serveur, de la servlet autonome à la plate-forme J2EE (*Java 2, Enterprise Edition*) complète, a été l'une des plus excitantes orientations de la programmation en Java. Le langage Java était à l'origine destiné à une utilisation dans de petits périphériques embarquables. Il a d'abord été utilisé pour le développement de contenu élaboré du côté client sous la forme d'applets. Mais jusqu'à ces dernières années, le potentiel de Java comme plate-forme de développement du côté serveur a été très largement négligé. Maintenant, Java est devenu un langage parfaitement adapté aux développements côté serveur.

Les entreprises, en particulier, ont très vite vu le potentiel de Java sur le serveur — Java est intrinsèquement adapté aux grandes applications client/serveur. La nature multi-plates-formes de Java est extrêmement utile aux entreprises qui ont un ensemble de serveurs hétérogènes exécutant différentes versions des systèmes d'exploitation Unix ou Windows (et de plus en plus Mac OS X). La conception de Java étant à la fois moderne et orientée objet, et offrant une protection de la mémoire, les développeurs peuvent réduire les coûts de développement et augmenter la robustesse. De plus, l'intégration dans Java du support réseau et des API d'entreprise offrent un accès aux données héritées, facilitant la migration depuis les anciens systèmes client/serveur.

Les servlets Java sont une composante clé du développement Java côté serveur. Une servlet est une petite extension intégrable à un serveur afin d'en étendre les fonctionnalités. Les servlets permettent au développeur d'étendre et de personnaliser tout serveur web ou tout serveur d'applications supportant Java avec un degré de portabilité, de souplesse et de facilité jusqu'ici inconnu. Mais avant d'aller plus en détail, mettons les choses en perspective.

Histoire des applications web

Bien que les servlets puissent être utilisées pour étendre les fonctionnalités des serveurs supportant Java, elles sont aujourd'hui le plus souvent utilisées pour étendre les serveurs web, offrant un remplacement puissant et efficace aux scripts CGI. Lorsque vous utilisez une servlet pour créer un contenu dynamique dans une page web ou pour étendre d'une autre façon les fonctionnalités d'un serveur web, en fait vous créez une appli-

cation web. Tandis qu'une page web ne fait qu'afficher un contenu statique et permettre à l'utilisateur de naviguer au travers de ce contenu, une application web offre une expérience plus interactive. Une application web peut être aussi simple qu'une recherche de mot-clé dans une archive ou aussi complexe que servir d'accès à une boutique électronique. Les applications web sont déployées sur l'Internet ou au sein d'intranets et d'extranets d'entreprise, où elles ont la possibilité d'améliorer la productivité et de changer la façon dont les sociétés, petites ou grandes, travaillent.

Pour comprendre la puissance des servlets, nous devons revenir en arrière et examiner les autres approches qui ont été utilisées pour créer des applications web.

Common Gateway Interface (CGI)

Common Gateway Interface, habituellement appelé CGI, a été l'une des premières techniques pratiques pour créer du contenu dynamique. Avec CGI, un serveur web envoie certaines requêtes à un programme extérieur. La sortie de ce programme est ensuite envoyée au client à la place d'un fichier statique. L'arrivée de CGI a permis d'implémenter toutes sortes de nouvelles fonctionnalités dans des pages web et CGI est vite devenu un standard *de facto*, implémenté sur des dizaines de serveurs web.

Il est intéressant de noter que la faculté des programmes CGI à créer des pages web dynamiques n'est qu'un effet secondaire de son but initial : définir une méthode standard pour qu'un serveur d'information communique avec des applications externes. Cela explique pourquoi CGI a peut-être le pire cycle de vie que l'on puisse imaginer. Lorsqu'un serveur reçoit une requête qui accède à un programme CGI, il doit créer un nouveau processus pour exécuter ce programme et ensuite lui passer, *via* des variables d'environnement et l'entrée standard, chaque bit d'information qui pourrait être nécessaire à la génération d'une réponse. La création d'un processus pour chacune de ces requêtes demande du temps et d'importantes ressources serveur, ce qui limite le nombre de requêtes qu'un serveur peut traiter de façon concurrente. La *figure 1-1* montre le cycle de vie d'un programme CGI.

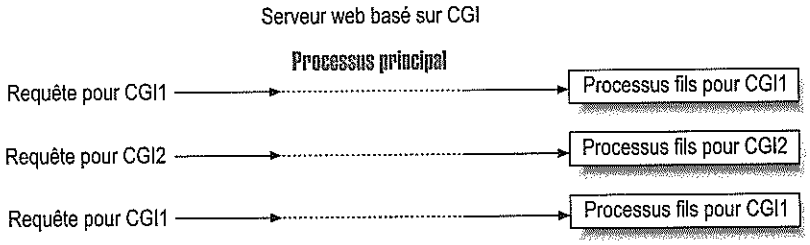


Figure 1-1. Cycle de vie de CGI

Même si un programme CGI peut être écrit en n'importe quel langage, Perl est devenu le choix prédominant. Ses possibilités de gestion de texte avancées sont d'une grande aide dans le traitement des détails de l'interface CGI. Écrire un script CGI en Perl lui donne un semblant d'indépendance de la plate-forme, mais il faut aussi que chaque requête démarre un interpréteur Perl séparé, ce qui prend encore plus de temps et demande encore plus de ressources.

Un autre problème souvent négligé est qu'un programme CGI ne peut pas interagir avec le serveur web pour tirer parti de ses possibilités une fois qu'il a commencé son exécution parce qu'il s'exécute dans un processus séparé. Par exemple, un script CGI ne peut pas écrire dans le fichier de journalisation du serveur. Pour plus d'informations sur la programmation CGI, consultez le livre *CGI Programming on the World Wide Web* de Shishir Gundavaram (O'Reilly & Associates).

FastCGI

Une société appelée Open Market a développé une alternative au standard CGI appelée FastCGI. Globalement, FastCGI fonctionne comme CGI. La différence importante est qu'un seul processus persistant est créé pour chaque programme FastCGI, comme montré en *figure 1-2*. Ceci élimine la nécessité de créer un nouveau processus pour chaque requête.

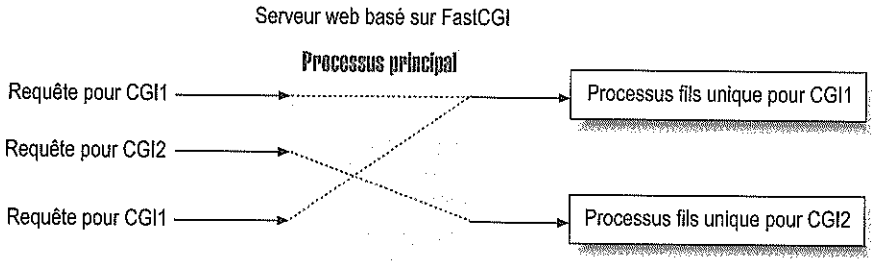


Figure 1-2. Cycle de vie de FastCGI

Bien que FastCGI soit un pas dans la bonne direction, le problème de la prolifération de processus n'est pas réglé : il y a au moins un processus pour chaque programme FastCGI. Si un programme FastCGI doit gérer des requêtes concurrentes, il a besoin d'une réserve de processus, un par requête. Si l'on considère que chaque processus peut exécuter un interpréteur Perl, cette approche n'est pas aussi extensible que l'on pourrait l'espérer. (Cependant, à son crédit, FastCGI peut distribuer ses processus sur de multiples serveurs.) Un autre problème avec FastCGI est qu'il n'aide en rien le programme FastCGI a interagir plus étroitement avec le serveur. Enfin, les programmes FastCGI sont aussi portables que le langage dans lequel ils ont été écrits. Pour plus d'informations sur FastCGI, visitez le site *http://www.fastcgi.com*.

PerlEx

PerlEx, développé par ActiveState, améliore les performances des scripts CGI écrits en Perl pour les serveurs web sous Windows NT (Internet Information Server de Microsoft, WebSite Professional de O'Reilly et FastTrack Server et Enterprise Server de iPlanet). Il a les mêmes avantages et inconvénients que FastCGI. Pour plus d'informations sur PerlEx, consultez le site *http://www.activestate.com/plex/*.

mod_perl

Si vous utilisez le serveur web Apache, une autre option pour améliorer les performances CGI est d'utiliser mod_perl. mod_perl est un module pour Apache qui intègre une

copie de l'interpréteur Perl dans l'exécutable Apache, fournissant un accès complet aux fonctionnalités de Perl dans le serveur. Les scripts CGI sont ainsi précompilés par le serveur et exécutés sans passer par une création de processus. Ils s'exécutent alors beaucoup plus vite et bien plus efficacement. L'inconvénient est que l'application doit obligatoirement être déployée sur le serveur Apache. Pour plus d'informations sur mod_perl, visitez le site *http://perl.apache.org/*.

Autres solutions

L'association CGI/Perl a l'avantage de produire du contenu web dynamique d'une façon plus ou moins indépendante de la plate-forme. D'autres technologies bien connues pour créer des applications web, telles que ASP et JavaScript côté du serveur, sont des solutions propriétaires qui ne fonctionnent qu'avec certains serveurs web.

API d'extension du serveur

Plusieurs sociétés ont créé des API propriétaires pour l'extension de leur serveur web. Par exemple, iPlanet/Netscape offre une API interne appelée WAI (précédemment NSAPI) et Microsoft fournit ISAPI. En utilisant l'une de ces API, vous pouvez écrire des extensions du serveur permettant d'en améliorer ou d'en changer les fonctionnalités de base, lui permettant ainsi de traiter des tâches qui étaient auparavant déléguées à des programmes CGI externes. Comme vous pouvez le voir dans la *figure 1-3*, les extensions du serveur existent à l'intérieur du processus principal du serveur Web.

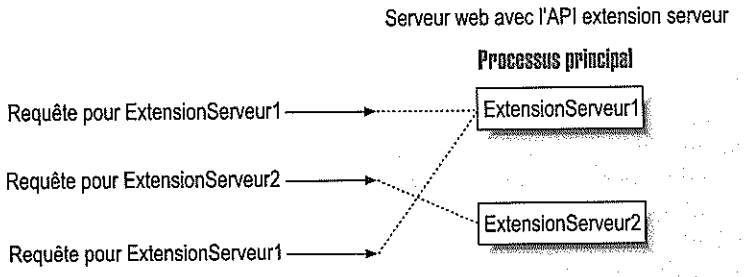


Figure 1-3. Cycle de vie d'une extension du serveur

Les API spécifiques à un serveur utilisant du code C ou C++, les extensions peuvent s'exécuter très rapidement et tirer pleinement parti des ressources du serveur. Cependant, elles ne sont pas une solution parfaite sur plusieurs plans. En dehors d'être difficiles à développer et à maintenir, elles posent d'importants problèmes de sécurité et de fiabilité : une extension du serveur qui se plante peut amener le serveur entier à s'arrêter ; une extension du serveur malveillante peut dérober les mots de passe des utilisateurs et créer des numéros de carte de crédit. Et, bien sûr, une extension propriétaire est inévitablement liée à l'API du serveur pour laquelle elle a été écrite, et bien souvent liée à un système d'exploitation particulier.

JavaScript côté serveur

iPlanet/Netscape a lui aussi une technique pour fournir les scripts sur le serveur, qu'il appelle *server-side JavaScript*, ou SSJS pour faire court. Comme ASP, SSJS permet d'inclure des morceaux de code dans des pages HTML pour générer du contenu web dynamique. La différence est que SSJS utilise JavaScript comme langage de scripts. Avec SSJS, les pages web sont précompilées afin d'améliorer les performances. Le support de SSJS n'est disponible qu'avec les serveurs de iPlanet/Netscape. Pour plus d'informations sur la programmation avec SSJS, reportez-vous à *http://developer.netscape.com/tech/javascript/ssjs/ssjs.html*.

Active Server Pages

Microsoft a développé une technique pour générer du contenu web dynamique appelée les *Active Server Pages*, ou ASP. Avec les ASP, une page HTML sur le serveur web peut contenir des morceaux de code embarqués (habituellement en VBScript ou Jscript, bien qu'il soit possible d'utiliser à peu près n'importe quel langage). Ce code est lu et exécuté par le serveur web avant qu'il n'envoie la page au client. Les ASP sont optimisées pour générer de petites portions de contenu dynamique à l'aide de composants COM pour les traitements lourds.

Le support des ASP est intégré à Microsoft Internet Information Server version 3.0 et supérieure, disponible gratuitement à *http://www.microsoft.com/iis*. Le support pour d'autres serveurs web est disponible sous forme de produit commercial chez Chili!Soft à l'adresse *http://www.chilisoft.com*. Sachez que les ASP s'exécutant sur une plate-forme non-Windows peuvent avoir des difficultés à réaliser des tâches avancées sans la bibliothèque COM de Windows. Pour plus d'informations sur la programmation des Active Server Pages, reportez-vous à *http://www.microsoft.com/asp* et à *http://www.activeserverpages.com/*.

JavaServer Pages

Les *JavaServer Pages*, appelées couramment *JSP*, sont une alternative Java aux ASP inventée et normalisée par Sun. Les JSP utilisent une syntaxe similaire aux ASP, mais le langage de script est Java. Contrairement aux ASP, les JSP sont une norme ouverte implémentée par des dizaines de fournisseurs sur toutes les plates-formes. Les JSP sont intimement liées aux servlets car une page JSP est transformée en une servlet pendant son exécution. Les JSP sont détaillées tout au long de ce livre. Pour plus d'informations sur les JSP, visitez le site *http://java.sun.com/products/jsp*.

Servlets Java

Bienvenue dans les servlets Java. Comme nous l'avons dit précédemment, une servlet est une extension de serveur générique — une classe Java qui peut être chargée dynamiquement afin d'étendre les fonctionnalités d'un serveur. Les servlets sont couramment utilisées sur les serveurs web, où elles peuvent prendre la place des scripts CGI. Une servlet est similaire à une extension de serveur propriétaire, excepté qu'elle s'exécute dans une machine virtuelle Java (JVM) sur le serveur (voir la *figure 1-4*), elle est donc sûre et portable. Les servlets opèrent uniquement à l'intérieur du domaine du serveur : contrairement aux applets, elles n'imposent pas le support de Java dans le navigateur web.

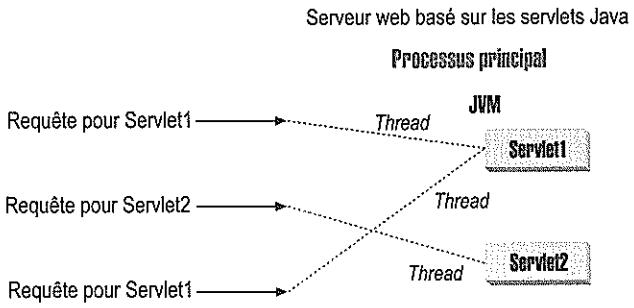


Figure 1-4. Cycle de vie d'une servlet

Contrairement à CGI et FastCGI, qui utilisent de multiples processus pour gérer des programmes séparés et/ou des requêtes séparées, les servlets peuvent être toutes gérées par des threads séparés au sein du même processus ou par des threads dans plusieurs processus répartis sur différents serveurs. Cela signifie aussi que les servlets sont efficaces et extensibles. Les servlets ayant un lien de communication bidirectionnel avec le serveur web, elles peuvent interagir très étroitement avec lui afin de réaliser des choses qui seraient impossibles avec des scripts CGI.

Un autre avantage des servlets est qu'elles sont portables : d'un système d'exploitation à l'autre, comme nous en avons l'habitude avec Java, mais aussi entre serveurs web. Comme nous le verrons bientôt, tous les serveurs web et les serveurs d'applications principaux supportent les servlets. Nous pensons que les servlets Java offrent la meilleure plate-forme possible pour le développement d'applications web, et nous en dirons beaucoup plus sur ce sujet plus tard dans ce chapitre.

Support des servlets

Comme Java lui-même, les servlets ont été conçues dans un but de portabilité. Les servlets sont supportées par toutes les plates-formes qui supportent Java et elles fonctionnent avec tous les principaux serveurs web[1]. Les servlets Java, telles que définies par la division Java Software de Sun Microsystems (connue sous le nom de JavaSoft), sont un *paquetage optionnel* à Java (appelé précédemment une *extension standard*). Cela signifie que les servlets sont officiellement reconnues par Sun et qu'elles font partie du langage Java, même si elles ne font pas partie de l'API noyau de Java. À la place, elles font maintenant partie intégrante de la plate-forme J2EE.

Pour faciliter le développement de servlets, Sun et Apache ont rendu disponibles les classes de l'API pour tout moteur web. Les paquetages `javax.servlet` et `javax.servlet.http` constituent cette API Servlet. La dernière version de ces classes est disponible

1. Notez que plusieurs fournisseurs de serveurs web ont leur propre implémentation de Java côté serveur, certaines ont aussi pris le nom de *servlets*. Elles sont généralement incompatibles avec les servlets Java telles que définies par Sun. La plupart de ces fournisseurs sont en train de convertir leur support Java vers les servlets standards ou sont en train d'introduire le support des servlets standards en parallèle, pour garantir une compatibilité ascendante.

en téléchargement à *http://java.sun.com/products/servlet/download.html*.[2] Tous les serveurs web supportant les servlets doivent utiliser ces classes en interne (même s'ils peuvent utiliser une autre implémentation). En général, ce fichier JAR peut donc aussi se trouver dans la distribution de votre serveur web compatible avec les servlets.

Peu importe où vous obtenez les classes pour les servlets, tant qu'elles sont disponibles sur votre système, puisque vous en avez besoin pour compiler vos servlets. En plus des classes pour les servlets, il vous faut également un système d'exécution des servlets (techniquement appelé un *conteneur de servlets*, et parfois un *moteur de servlets*) afin de pouvoir les tester et les déployer. Le choix du conteneur de servlets dépend en partie du ou des serveurs web que vous utilisez. Il existe trois déclinaisons de conteneurs de servlets : *autonome*, *additionnel* et *embarquable*.

Conteneurs de servlets autonomes

Un conteneur de servlets autonome est un serveur qui intègre le support des servlets. Un tel conteneur a l'avantage que tout fonctionne directement. Cependant, l'inconvénient est que vous devez attendre une nouvelle version du serveur web pour obtenir le support de la dernière version des servlets. Un autre inconvénient est que les fournisseurs de serveur supportent en général que la JVM du fournisseur. Voici les serveurs web incluant un conteneur autonome.

- Tomcat Server d'Apache constitue l'implémentation de référence officielle du support des servlets par un conteneur de servlets. Il est écrit entièrement en Java et il est disponible gratuitement sous forme d'une licence open source. Tout le code source est disponible et tout le monde peut participer à son développement. Ce serveur peut fonctionne en mode autonome ou en mode additionnel apportant le support des servlets à Apache ou à d'autres serveurs. Il peut même fonctionner comme conteneur embarqué. À côté de Tomcat, Apache développe l'implémentation standard des paquetages `javax.servlet` et `javax.servlet.http`. Au moment de l'écriture de ce livre, les servlets sont les seuls paquetages `java.*` ou `javax.*` maintenus officiellement en *open source*.[3] Visitez *http://jakarta.apache.org*.
- Web Server Enterprise Edition (version 4.0 et ultérieure) de iPlanet (Netscape) est peut-être le serveur web le plus connu offrant un support intégré des servlets. Certains tests de performance montrent que ce serveur dispose de l'implémentation des servlets la plus rapide. À côté de cela, même si les versions 3.51 et 3.6 intègre le support des servlets, ils n'acceptent que l'API Servlet 1.0 et souffrent d'un nombre de bogues si important que les servlets sont pratiquement inutilisables. Pour utiliser les servlets avec les serveurs Netscape 3.x, il est préférable d'utiliser un conteneur de servlets additionnel. Voir *http://www.iplanet.com*.

---

2. À un moment donné il était prévu que ces classes soient intégrées au JDK 1.2. Cependant, il fut décidé plus tard de garder les classes des servlets séparées du JDK afin de faciliter les révisions et les corrections de l'API Servlet.

3. Le fait d'avoir une implémentation en *open source* des paquetages `javax.servlet` et `javax.servlet.http` a permis de corriger de nombreux bogues (par exemple, Jason a modifié `HttpServlet` afin d'améliorer le comportement d'une requête GET conditionnelle) sans problèmes de compatibilité. Nous espérons que cela va encourager la sortie en *open source* des paquetages Java plus officiels.

- WebSite Professional de O'Reilly a des fonctionnalités similaires à Enterprise Server d'iPlanet pour un prix plus faible. Voir *http://website.oreilly.com*.
- Zeus Web Server est un serveur web que certains considèrent comme le plus rapide. Sa liste de fonctionnalités est plutôt longue et inclut le support des servlets. Voir *http://www.zeus.co.uk*.
- Resin de Caucho, un conteneur en *open source* se vante de ses performances. Il peut fonctionner en mode autonome et comme conteneur additionnel pour beaucoup de serveurs. Voir *http://www.caucho.com*.
- LiteWebServer de Gefion Software est un petit (un peu plus de 100 K) conteneur de servlets destiné à des utilisations dans lesquelles une faible taille est importante, par exemple pour des démonstrations. Voir *http://www.gefionsoftware.com/LiteWebServer*.
- Jigsaw Server du World Wide Web Consortium est disponible en *open source* et il est écrit entièrement en Java. Voir *http://www.w3.org/Jigsaw*.
- Java Web Server de Sun est le serveur par qui tout a commencé. Il a été le premier serveur à implémenter les servlets et il a servi d'implémentation de référence pour l'API Servlet 2.0. Il est écrit entièrement en Java (excepté deux bibliothèques de code natif qui étendent ses fonctionnalités mais qui ne sont pas indispensables). Sun a interrompu les développements de ce serveur, pour se concentrer sur les produits iPlanet/Netscape de l'alliance Sun-Netscape. Voir *http://java.sun.com/products*.

Les serveurs d'applications sont un domaine de développement fertile. Un serveur d'applications offre, sur le serveur, le support pour le développement d'applications d'entreprise. La plupart des serveurs d'applications basés sur Java supportent les servlets et le reste de la spécification J2EE. Parmi ces serveurs, on trouve :

- WebLogic Application Server de BEA System, un des premiers et des plus connus serveurs d'applications basés sur Java. Voir *http://www.beasys.com/products/weblogic*.
- Orion Application Server, un serveur haut de gamme mais relativement peu coûteux écrit entièrement en Java. Voir *http://www.orionserver.com*.
- Enhydra Application Server, un serveur en *open source* de Lutris. Voir *http://www.enhydra.org*.
- Borland Application Server 4, une serveur ayant une orientation spéciale sur CORBA. Voir *http://www.borland.com/appserver*.
- WebSphere Application Server d'IBM, un serveur haut de gamme basé en partie sur le code d'Apache. Voir *http://www-4.ibm.com/software/webservers*.
- Dynamo Application Server 3 de ATG, un autre serveur haut de gamme écrit entièrement en Java. Voir *http://www.atg.com*.
- Application Server d'Oracle, un serveur conçu pour une intégration avec la base de données Oracle. Voir *http://www.oracle.com/appserver*.
- iPlanet Application Server, le grand frère compatible J2EE de iPlanet Web Server Enterprise Edition. Voir *http://www.iplanet.com/products/infrastructure/app_servers/nas*.
- GemStone/J Application Server, un serveur Java d'une société connue précédemment pour son serveur Smalltalk. Voir *http://www.gemstone.com/products/j*.

- JRun Server d'Allaire (précédemment de Live Software), un conteneur de servlets simple qui a grandi pour devenir un conteneur avancé fournissant plusieurs des technologies J2EE dont les EJB, JTA et JMS. Voir *http://www.allaire.com/products/jrun*.
- Silverstream Application Server, un serveur totalement compatible avec J2EE qui a aussi débuté avec les servlets. Voir *http://www.silverstream.com*.

Conteneurs de servlets additionnels

Un conteneur de servlets additionnel fonctionne comme un *plug-in* à un serveur existant. Il ajoute le support des servlets à un serveur qui n'a pas été conçu à l'origine pour cela ou à un serveur ayant une implémentation des servlets ancienne ou médiocre. Des conteneurs de servlets additionnels ont été écrits pour plusieurs serveurs y compris Apache, FastTrack Server et Enterprise Server d'iPlanet, Internet Information Server et Personal Web Server de Microsoft, WebSite d'O'Reilly, Go Webserver de Lotus Domino, WebSTAR de StarNine et AppleShare IP d'Apple. Voici une liste de conteneurs de servlets additionnels :

- ServletExec de New Atlanta, un plug-in conçu pour supporter les servlets sur tous les serveurs web populaires sur tous les systèmes d'exploitation populaires. Il inclut un débogueur gratuit. Voir *http://www.servletexec.com*.
- JRun d'Allaire (précédemment de Live Software), est disponible sous forme de plug-in pour apporter les servlets à tous les serveurs web populaires sur tous les systèmes d'exploitation populaires. Voir *http://www.allaire.com/products/jrun/*.
- Le module Jserv du projet Java-Apache, un conteneur de servlets disponible gratuitement en *open source* qui ajoute le support des servlets au très fameux serveur Apache. Les développements sur JServ se sont arrêtés et Tomcat Server (agissant comme un plug-in) en est le remplaçant. Voir *http://java.apache.org/*.
- Tomcat Server d'Apache, comme on vient de le présenter. Tomcat peut être connecté à d'autres serveurs dont Apache, iPlanet/Netscape et IIS.

Conteneurs de servlets embarquables

Un conteneur embarquable est généralement une plate-forme de déploiement de servlets légère qui peut être embarquée dans une autre application. Cette application devient alors un vrai serveur. Voici des exemples de moteurs de servlets embarquables :

- Tomcat Server d'Apache, bien que généralement utilisé en mode autonome ou additionnel, peut aussi être embarqué dans une autre application. Ce serveur étant disponible en *open source*, le développement de la plupart des autres conteneurs embarquables s'est arrêté.
- Nexus Web Server d'Anders Kristensen, un logiciel disponible gratuitement permettant d'exécuter les servlets. Il implémente la majorité de l'API Servlet et il peut être facilement embarqué dans des applications Java. Voir *http://www-uk.hpl.hp.com/people/ak/java/nexus/*.

Autres aspects

Avant d'aller plus loin, nous nous sentons obligés de souligner que tous les conteneurs de servlets ne sont pas égaux. Ainsi, avant de choisir un conteneur de servlets (et peut-être un serveur) avec lequel déployer vos servlets, n'hésitez pas à le tester. Poussez-le vers ses limites. Regardez les listes de diffusion. Vérifiez toujours que vos servlets se comportent de la même façon qu'avec l'implémentation de référence Tomcat. Vous pouvez aussi vérifier que des outils de développement sont fournis, quelles technologies J2EE sont supportées et quels sont les délais de réponse du support en ligne. L'intérêt des servlets, c'est qu'il est inutile de s'inquiéter du plus petit dénominateur commun sur l'implémentation, vous pouvez donc choisir le conteneur de servlets qui possède les fonctionnalités que vous voulez.

Pour une liste complète et mise à jour des conteneurs de servlets disponibles, avec les informations de prix, consultez le site *http://www.servlets.com*.

Puissance des servlets

Jusqu'à présent, nous avons montré les servlets comme une alternative à d'autres technologies permettant d'avoir un contenu web dynamique, mais nous n'avons pas réellement expliqué pourquoi nous pensons que vous devriez les utilisez. Qu'est-ce qui fait des servlets un choix viable pour le développement web ? Nous pensons que les servlets offrent de nombreux avantages par rapport aux autres approches : portabilité, puissance, efficacité, endurance, sûreté, élégance, intégration, extensibilité et souplesse. Examinons-les une à une.

Portabilité

Les servlets étant écrites en Java et se conformant à une API bien définie et largement acceptée, elles ont de grandes chances d'être portables sur plusieurs systèmes d'exploitation et sur plusieurs implémentations de serveur. Vous pouvez développer une servlet sur une machine Windows NT exécutant le serveur Tomcat et ensuite la déployer sans effort sur un serveur UNIX haut de gamme exécutant iPlanet/Netscape Application Server. Avec les servlets, vous pouvez réellement « écrire une fois, servir partout ».

La portabilité des servlets n'est pas la pierre d'achoppement que l'on a si souvent avec les applets, et cela pour deux raisons. Premièrement, la portabilité des servlets n'est pas obligatoire. Contrairement aux applets, qui doivent être testées sur toutes les plates-formes client possibles, les servlets n'ont besoin de fonctionner que sur la machine serveur que vous utilisez pour le développement et le déploiement. À moins que votre métier ne soit de vendre vos servlets, il est inutile de s'occuper de leur totale portabilité. Deuxièmement, les servlets évitent la portion du langage Java la plus sujette à erreurs et la moins cohérente : l'AWT (*Abstract Windowing Toolkit*) qui forme les bases des interfaces utilisateur graphiques de Java, y compris Swing.

Puissance

Les servlets peuvent bénéficier de la pleine puissance des API noyau de Java : réseau et accès URL, multithread, manipulation d'images, compression de données, connectivité

aux bases de données (JDBC), internationalisation, invocation de méthodes distantes (*Remote Method Invocation* — RMI) et intégration aux systèmes hérités (CORBA). Les servlets peuvent aussi exploiter la plate-forme J2EE qui inclut le support des Enterprise JavaBeans (EJB), des transactions distribuées (JTS), de la messagerie normalisée (JMS), de la recherche dans les annuaires (JNDI) et les accès avancés aux bases de données (JDBC 2.0). La liste des API standard disponibles aux servlets continue de grandir, rendant la tâche de développement des applications web plus rapide, plus facile et plus fiable.

En tant qu'auteur de servlets, vous pouvez aussi choisir dans une pléthore de classes Java et de composants JavaBeans disponibles sur le marché. Les servlets peuvent utiliser du code provenant d'un tiers pour gérer des tâches comme la recherche d'expressions régulières, les graphiques de données, l'accès personnalisé aux bases de données, l'utilisation avancée du réseau, l'analyse XML syntaxique et les traductions XSLT.

Les servlets sont aussi bien adaptées aux communications client/serveur. Avec une applet Java et une servlet Java, vous pouvez utiliser RMI et la sérialisation d'objets pour gérer la communication client/serveur. Cela signifie que le même code personnalisé peut se trouver sur le client et sur le serveur. Réaliser la même chose sur le serveur avec des langages autres que Java est bien plus compliqué, car il faut développer votre propre protocole personnalisé pour gérer la communication.

Efficacité et endurance

L'invocation d'une servlet est très efficace. Une fois la servlet chargée, elle reste généralement dans la mémoire du serveur comme une seule instance d'objet. Par la suite, le serveur invoque la servlet pour traiter une requête en utilisant une invocation de méthode simple et légère. Contrairement à CGI, il n'y a pas de processus à créer ou d'interpréteur à invoquer. La servlet peut donc commencer à traiter la requête presque immédiatement. Des requêtes multiples et concurrentes sont traitées par des threads séparés. Les servlets supportent donc très bien l'extension du système.

En général, les servlets sont des objets naturellement endurants. Une servlet restant dans la mémoire du serveur comme une seule instance d'objet, elle maintient automatiquement son état et elle peut conserver des références à des ressources externes, comme des connexions à des bases de données, qui sinon seraient longues à rétablir.

Sûreté

Les servlets supportent des pratiques de programmation sûre à plusieurs niveaux. Parce qu'elles sont écrites en Java, les servlets héritent du typage fort de ce langage. De plus, l'API Servlet est implémentée pour être sûre en ce qui concerne les types. Tandis que la plupart des valeurs dans un programme CGI, y compris une valeur numérique comme le numéro de port d'un serveur, sont traitées comme des chaînes de caractères, les valeurs sont manipulées par l'API Servlet en utilisant leur type natif. Un numéro de port de serveur est donc représenté comme un entier. Le ramasse-miettes automatique de Java et l'absence de pointeur signifie que les servlets sont généralement sûres en ce qui concerne les problèmes de gestion de la mémoire tels que les pointeurs sur un espace mémoire vide, les références invalides et les fuites de mémoire.

Les servlets peuvent gérer les erreurs de façon sûre, grâce au mécanisme de gestion des exceptions de Java. Si une servlet effectue une division par zéro ou effectue toute autre

opération illégale, elle lance une exception qui peut être attrapée et gérée de façon sûre par le serveur. Celui-ci peut alors enregistrer l'erreur et poliment s'excuser auprès de l'utilisateur. Si une extension du serveur écrite en C++ effectuait la même erreur, elle planterait certainement le serveur.

Un serveur peut aussi se protéger lui-même des servlets par l'utilisation du gestionnaire de sécurité de Java. Il peut exécuter ses servlets sous le contrôle d'un gestionnaire de sécurité strict qui, par exemple, garantit une politique de sécurité conçue pour empêcher qu'une servlet malveillante ou mal écrite n'endommage son système de fichiers.

Élégance

L'élégance du code d'une servlet est frappante. Il est propre, orienté objet, modulaire et étonnamment simple. Une des raisons de cette simplicité réside dans l'API Servlet elle-même, qui inclut des méthodes et des classes pour gérer les tâches routinières du développement d'une servlet. Même les opérations avancées, comme la gestion des cookies et le suivi de session, sont prises en charge par des classes utilitaires. Quelques autres tâches avancées mais courantes n'ont pas été incluses dans l'API, et, pour les traiter, nous avons donc essayé de fournir un ensemble de classes utiles dans le paquetage `com.oreilly.servlet`.

Intégration

Les servlets sont très étroitement intégrées au serveur. Cette intégration permet à une servlet de coopérer avec le serveur d'une façon impossible à un programme CGI. Par exemple, une servlet peut utiliser le serveur pour convertir des chemins de fichier, effectuer une journalisation, vérifier les autorisations et effectuer une correspondance de type MIME. Des extensions spécifiques au serveur pourraient faire beaucoup plus, mais le processus est généralement plus complexe et source d'erreurs.

Extensibilité et souplesse

L'API Servlet est conçue pour être facilement étendue. Aujourd'hui, cette API inclut des classes optimisées pour les servlets HTTP. Mais dans le futur, elle pourra être étendue et optimisée pour un autre type de servlets, soit par Sun, soit par quelqu'un d'autre. Il est aussi possible que son support pour les servlets HTTP puisse être étendu.

Les servlets sont aussi assez souples dans la création de contenu. Elles peuvent générer du contenu simple avec des instructions `out.println()` ou des pages complexes à l'aide d'un moteur de template. Elles peuvent créer une page HTML en traitant la page comme un ensemble d'objets Java ou en effectuant une transformation XML vers HTML. Les servlets peuvent même servir à construire de nouvelles technologies comme les Java-Server Pages. Qui sait ce qu'ils vont (ou ce que vous allez) nous concocter pour la prochaine fois.

2

Introduction aux servlets HTTP

Ce chapitre contient un court tutoriel sur l'écriture et l'exécution d'une servlet HTTP simple. Il explique ensuite comment déployer la servlet dans une application web standard et comment configurer son comportement avec un descripteur de déploiement XML.

Contrairement à la première édition, ce chapitre ne concerne pas les SSI (*server-side includes*) ni le chaînage ou le filtrage de servlets. En effet ces techniques, aussi utiles qu'elles ont été et en dépit du fait qu'elles ont été implémentées dans Java Web Server, n'ont jamais été validées par la spécification des servlets (qui est sortie après celle de la première édition de ce livre). Les SSI ont été remplacée par de nouvelles techniques d'inclusion par programmation. Le chaînage de servlets a été jugé trop inélégant pour être approuvé officiellement, bien que l'idée de base semble réapparaître dans l'API Servlet 2.3 sous la forme d'un mécanisme général de pré- et post-filtrage.

Bien que le code de chaque exemple de ce chapitre et du reste du livre soit disponible en téléchargement tant sous sa forme source que compilée (comme le décrit la préface), nous vous suggérons que, au moins pour ce premier chapitre, vous n'utilisiez pas Internet et que vous entriez vous-même les exemples. Cela devrait vous aider à mieux assimiler les concepts. Ne soyez pas alarmé si vous pensez que nous survolons certains sujets dans ce chapitre. Les servlets sont puissantes et, certaines fois, complexes. Nous souhaitons donner ici une vue générale du fonctionnement avant d'aller au cœur des choses et de vous ensevelir sous les détails. À la fin de ce livre, nous promettons que vous serez capable d'écrire des servlets qui font absolument tout, sauf peut-être le café.

Bases de HTTP

Avant même de pouvoir vous montrer une servlet HTTP simple, nous devons nous assurer que vous avez au moins une compréhension minimale du fonctionnement du protocole derrière le web : HTTP. Si vous êtes un programmeur CGI expérimenté (ou si vous avez réalisé des programmes web du côté du serveur importants), vous pouvez sauter cette section. Mais mieux encore, vous pourriez en profiter pour rafraîchir votre mé-

moire sur les points de détail des méthodes GET et POST. Cependant, si vous êtes un nouveau venu dans le monde de la programmation web du côté du serveur, vous devez lire attentivement cette section, car le reste du livre suppose que vous compreniez HTTP. Pour des explications plus détaillées sur HTTP et ses méthodes, voir *Programmation de clients Web en Perl* de Clinton Wong (Éditions O'Reilly).

Requêtes, réponse et en-têtes

HTTP est un protocole simple, sans état. Un client, comme un navigateur web, crée une requête, le serveur répond et la transaction est terminée. Lorsque le client envoie une requête, la première chose qu'il spécifie est une commande HTTP, appelée une *méthode*, qui indique au serveur le type d'action à effectuer. La première ligne de la requête spécifie aussi l'adresse d'un document (une URL) et la version du protocole HTTP qu'il utilise. Par exemple :

```
GET /intro.html HTTP/1.0
```

Cette requête utilise la méthode GET pour demander le document appelé *intro.html*, en utilisant HTTP version 1.0. Après avoir envoyé la requête, le client peut envoyer des en-têtes d'information optionnels pour passer au serveur des informations complémentaires sur la requête, telles que le logiciel utilisé par le client et le type de contenu qu'il comprend. Cette information n'est pas directement liée à la requête, mais elle peut être utilisée par le serveur pour générer sa réponse. Voici quelques exemples d'en-tête de requêtes :

```
User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: image/gif, image/jpeg, text/*, */*
```

L'en-tête `User-Agent` fournit une information sur le logiciel du client, tandis que l'en-tête `Accept` spécifie les types de contenu (MIME) que le client préfère recevoir. (Nous en dirons plus sur les en-têtes de requêtes dans le contexte des servlets au chapitre 4, *Retrouver l'information*. Après les en-têtes, le client envoie une ligne vide pour indiquer la fin de la section d'en-tête. Le client peut aussi envoyer des données supplémentaires, si cela est approprié à la méthode utilisée, comme c'est le cas avec la méthode POST dont nous parlerons bientôt. Si la requête n'envoie pas de données, elle se termine par une ligne vide.

Après que le client ait envoyé la requête, le serveur la traite et renvoie une réponse. La première ligne de la réponse est une ligne d'état qui spécifie la version du protocole HTTP utilisé par le serveur, un code d'état et une description du code d'état. Par exemple :

```
HTTP/1.0 200 OK
```

Cette ligne d'état inclut le code d'état 200 qui indique que la requête a été traitée avec succès, d'où la description OK. Un autre code d'état très courant est 404, avec la description Not Found — qui comme vous l'avez deviné, signifie que le document demandé n'a pas été trouvé. Le chapitre 5, *Envoyer une information HTML*, explique les différents codes d'état courants et comment vous pouvez les utiliser dans les servlets. L'annexe C, *Descripteur de déploiement, référence de la DTD*, donne une liste complète des codes d'état HTTP.

Après la ligne d'état, le serveur envoie des en-têtes de réponse donnant des informations

client comme le logiciel utilisé par le serveur et le type de contenu de la réponse du serveur. Par exemple :

```
Date: Saturday, 23-May-00 03:25:12 GMT
Server: Tomcat Web Server/3.2
MIME-version: 1.0
Content-type: text/html
Content-length: 1029
Last-modified: Thursday, 7-May-00 12:15:35 GMT
```

L'en-tête `Server` fournit une information sur le logiciel du serveur, tandis que l'en-tête `Content-type` spécifie le type MIME des données contenues dans la réponse. (Dans le chapitre 5, nous parlerons aussi des en-têtes de réponse.) Le serveur envoie une ligne vide après les en-têtes pour conclure la section d'en-tête.

Si la requête a été traitée avec succès, les données demandées sont ensuite envoyées dans la réponse. Sinon, la réponse peut contenir des données lisibles par un humain indiquant pourquoi le serveur n'a pu satisfaire la requête.

GET et POST

Lorsqu'un client se connecte à un serveur et envoie une requête HTTP, cette requête peut être de plusieurs types différents, appelés méthodes. Les méthodes les plus fréquemment utilisées sont GET et POST. Pour simplifier, la méthode GET est conçue pour obtenir des informations (document, graphique, ou le résultat d'une requête sur une base de données), tandis que la méthode POST est conçue pour poster des informations (un numéro de carte de crédit, de nouvelles données graphiques, ou des informations à stocker dans une base de données). Si l'on prend l'analogie du tableau d'affichage, GET sert à lire et POST à afficher un nouveau message. GET est la méthode utilisée lorsque vous entrez une URL directement dans votre navigateur ou lorsque vous cliquez sur un lien ; GET ou POST sont utilisées pour l'envoi d'un formulaire HTML.

La méthode GET, bien qu'elle soit conçue pour obtenir des informations, peut inclure dans la requête ses propres informations qui décrivent mieux ce qu'elle veut recevoir, par exemple une échelle en x et y pour un graphique créé dynamiquement. Cette information est passée sous forme d'une séquence de caractères ajoutés à l'URL de la requête dans ce que l'on appelle une *chaîne d'interrogation*. En plaçant des informations supplémentaires de cette façon dans l'URL, la page peut être ajoutée aux signets ou envoyée par courrier électronique comme n'importe quelle autre. Les requêtes GET n'étant pas supposées, en théorie, envoyer de grandes quantités d'informations, certains serveurs limitent la longueur des URL et des chaînes d'interrogation à 240 caractères.

La méthode POST utilise une technique différente pour envoyer des informations au serveur car, dans certains cas, on peut avoir besoin d'envoyer des méga-octets d'informations. Une requête POST passe toutes ses données, d'une taille illimitée, directement sur la connexion par socket dans le corps de la requête HTTP. L'échange est invisible au client. L'URL ne change pas. Par conséquent, les requêtes POST ne peuvent pas être ajoutées aux signets ou envoyées par courrier électronique, ni même, dans certains cas, rechargées. Cela a été prévu au moment de la conception ; des informations envoyées au serveur, comme le numéro de votre carte de crédit, ne doivent être envoyées qu'une

seule fois. POST apporte aussi une sécurité supplémentaire lors des envois d'informations sensibles car le journal du serveur qui enregistre tous les accès aux URL ne contient pas les données POST envoyées.

En pratique, les utilisations de GET et POST se sont éloignées des objectifs initiaux. Il est classique d'utiliser POST au lieu de GET pour envoyer des demandes d'information ayant des paramètres de grande taille afin d'éviter les problèmes liés aux URL trop longues. Il est aussi courant que des formulaires simples qui envoient de l'information utilisent GET, parce que (pourquoi pas) ça marche ! En général, ce n'est pas un problème. Souvenez-vous simplement que les requêtes GET, bien qu'elles puissent être facilement ajoutées aux signets, ne doivent pas permettre de provoquer des dommages pour lesquels le client pourrait être tenu pour responsable. Autrement dit, les requêtes GET ne doivent pas être utilisées pour envoyer une demande, mettre à jour une base de données, ou entreprendre une action explicite de quelque façon que ce soit.

Autres méthodes

En plus de GET et POST, il existe d'autres méthodes HTTP moins utilisées. Il existe la méthode HEAD qui est envoyée par un client lorsqu'il veut seulement voir les en-têtes de la réponse afin de déterminer la taille des documents, l'heure de modification, ou une disponibilité générale. Il y a aussi PUT, pour placer des documents directement sur le serveur, et DELETE, pour l'action opposée. Ces deux dernières ne sont pas largement supportées du fait des problèmes de sécurité complexes qu'elles apportent. La méthode TRACE est utilisée comme une aide au débogage — elle retourne au client le contenu exact de la demande. Enfin, la méthode OPTIONS peut être utilisée pour demander au serveur quelles méthodes il supporte et quelles sont les options disponibles pour une ressource particulière sur le serveur.

L'API Servlet

Maintenant que vous possédez une connaissance minimale de HTTP, nous pouvons continuer et parler de l'API Servlet que nous allons utiliser pour créer des servlets HTTP, ou toute autre sorte de servlets. Les servlets utilisent des classes et des interfaces issues de deux paquetages : `javax.servlet` et `javax.servlet.http`. Le paquetage `javax.servlet` contient les classes pour un support des servlets génériques et indépendantes du protocole. Ces classes sont étendues par celles du paquetage `javax.servlet.http` qui ajoute des fonctionnalités spécifiques à HTTP. Le nom du paquetage du niveau le plus haut est `javax` au lieu du `java` plus familier, pour indiquer que l'API Servlet est *paquetage optionnel* (appelé précédemment une *extension standard*).

Chaque servlet doit implémenter l'interface `javax.servlet.Servlet`. La plupart des servlets l'implémente en étendant l'une des deux classes spéciales : `javax.servlet.GenericServlet` ou `javax.servlet.http.HttpServlet`. Une servlet indépendante du protocole doit être une sous-classe de `GenericServlet`, tandis qu'une servlet HTTP doit être une sous-classe de `HttpServlet`, elle-même une sous-classe de `GenericServlet` à laquelle on a ajouté des fonctionnalités spécifiques à HTTP.

Contrairement à un programme Java classique, mais comme une applet, une servlet n'a pas de méthode `main()`. À la place, certaines méthodes d'une servlet sont invoquées par

le serveur lors du processus de gestion des requêtes. Chaque fois que le serveur achemine une requête à une servlet, il invoque sa méthode `service()`.

Une servlet générique doit surcharger sa méthode `service()` afin de traiter les requêtes de façon appropriée. La méthode `service()` accepte deux paramètres : un objet de requête et un objet de réponse. L'objet de requête informe la servlet sur la requête, tandis que l'objet de réponse est utilisé pour retourner une réponse. La *figure 2-1* montre comment une servlet générique traite les requêtes.

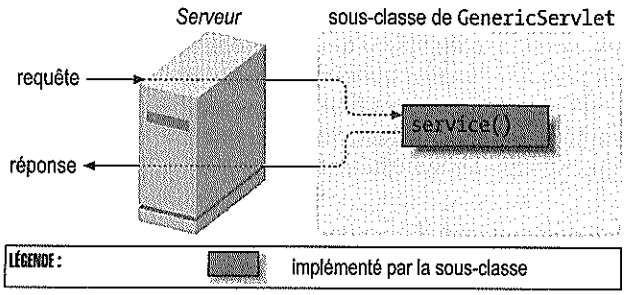


Figure 2-1. Servlet générique traitant une requête

À l'opposé, une servlet HTTP ne surcharge pas habituellement la méthode `service()`. Au lieu de cela, elle surcharge `doGet()` pour gérer les requêtes GET et `doPost()` pour les requêtes POST. Une servlet HTTP peut surcharger l'une ou l'autre de ces méthodes, en fonction du type de requête qu'elle va traiter. La méthode `service()` de `HttpServlet` prend en charge l'initialisation et l'acheminement de toutes les méthode do*XXX*(), ce qui explique pourquoi elle n'est habituellement pas surchargée. La *figure 2-2* montre comment une servlet HTTP gère les requêtes GET et POST.

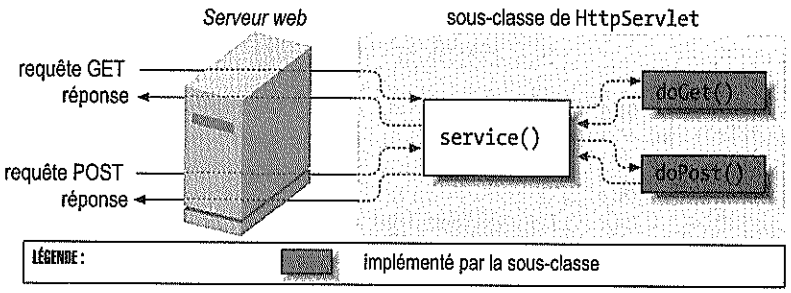


Figure 2-2. Servlet HTTP traitant les requêtes GET et POST

Une servlet HTTP peut surcharger les méthodes `doPut()` et `doDelete()` pour gérer les requêtes PUT et DELETE, respectivement. Cependant les servlets HTTP ne touchent généralement pas à `doTrace()`, ou `doOptions()`. Pour ces méthodes, les implémentations par défaut sont généralement suffisantes.

Le reste des paquetages `javax.servlet` et `javax.servlet.http` est principalement constitué de classes support. Par exemple, les classes `ServletRequest` et `ServletResponse` dans `javax.servlet` offrent un accès à des requêtes et des réponses génériques du ser-

veur, tandis que `HttpServletRequest` et `HttpServletResponse` dans `javax.servlet.http` offrent un accès à des requêtes et des réponses HTTP. Le paquetage `javax.servlet.http` contient aussi une interface `HttpSession` qui offre une fonctionnalité de suivi de session intégrée et une classe `Cookie` permettant d'établir et de traiter rapidement des cookies HTTP.

Générer des pages

La servlet HTTP la plus basique est capable de générer une page HTML complète. Cette servlet a accès aux mêmes informations que celles habituellement envoyées à un script CGI, voire un peu plus. Une servlet qui génère une page HTML peut être utilisée pour toutes les tâches où l'on utilise souvent CGI, par exemple pour le traitement des formulaires HTML, pour la production de rapports venant d'une base de données, pour prendre des ordres, pour vérifier les identités, etc.

Écrire « Bonjour tout le monde »

L'*exemple 2-1* montre une servlet HTTP qui génère une page HTML complète. Pour faire aussi simple que possible, cette servlet écrit juste « Bonjour tout le monde » chaque fois qu'un navigateur web y accède[1].

Exemple 2-1. Servlet affichant « Bonjour tout le monde »

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>Bonjour tout le monde</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<BIG>Bonjour tout le monde</BIG>");
    out.println("</BODY></HTML>");
  }
}
```

1. La première fois qu'un programme « Bonjour tout le monde » documenté est apparu c'était dans *A Tutorial Introduction to the Language B*, écrit par Brian Kernighan en 1973. Pour ceux qui sont trop jeunes pour s'en rappeler, B était un précurseur de C. Vous pourrez trouver plus d'informations sur le langage de programmation B et un lien au tutoriel à *http://cm.bell-labs.com/who/dmr/bintro.html*.

Cette servlet étend la classe `HttpServlet` et surcharge la méthode `doGet()`. Chaque fois que le serveur web reçoit une requête GET pour cette servlet, il invoque cette méthode `doGet()`, en lui passant un objet `HttpServletRequest` et un objet `HttpServletResponse`.

`HttpServletRequest` représente la requête du client. Cet objet donne à la servlet un accès aux informations sur le client, aux paramètres de la requête, à l'en-tête HTTP passé avec la requête, et ainsi de suite. Le chapitre 4 explique toutes les possibilités de l'objet de requête. Dans notre exemple, nous l'avons complètement ignoré. Après tout, cette servlet écrira toujours « Bonjour tout le monde » quelle que soit la requête !

`HttpServletResponse` représente la réponse de la servlet. Une servlet peut utiliser cet objet pour retourner des données au client. Ces données peuvent être de n'importe quel type, bien que ce type doive être spécifié dans la réponse. Une servlet peut aussi utiliser cet objet pour envoyer des en-têtes de réponse HTTP. Les chapitres 5 et 6 expliquent tout ce qu'une servlet peut faire dans la réponse.

Notre servlet commence par utiliser la méthode `setContentType()` de l'objet de réponse pour indiquer que le contenu de la réponse sera du type `text/html`, le type MIME standard pour le contenu de pages HTML. Ensuite, elle utilise la méthode `getWriter()` pour retrouver un `PrintWriter`, l'équivalent international d'un `PrintStream`. `PrintWriter` convertit les caractères Unicode de Java en un codage spécifique à la *locale*. Pour une locale anglaise, il se comporte comme `PrintStream`. Enfin, la servlet utilise ce `PrintWriter` pour envoyer le texte « Bonjour tout le monde » au client.

Et voilà ! C'est tout ce que le code doit faire pour dire bonjour à tous ceux qui « surfent » sur notre servlet.

Exécuter la servlet HelloWorld

Lorsque vous développez des servlets, vous avez besoin de deux choses : les fichiers des classes de l'API Servlet, utilisés pour la compilation, et un conteneur de servlets, comme un serveur web, utilisé pour le déploiement. Tous les conteneurs de servlets connus fournissent les fichiers de classe de l'API Servlet. Les deux contraintes sont donc satisfaites par un seul téléchargement.

Il existe des dizaines de conteneurs de servlets disponibles pour le déploiement d'une servlet, dont plusieurs ont été listés dans le chapitre 1, *Introduction*. Assurez-vous simplement que le serveur choisi support la version 2.2, ou ultérieure, de l'API Servlet. Elle a été la première version de l'API à supporter les applications web telles que présentées dans ce chapitre. Une liste des conteneurs de servlet existants et le niveau de l'API qu'il supporte est disponible sur le site *http://www.servlets.com*.

Bon, que faisons-nous de notre code afin qu'il tourne dans un serveur web ? Et bien, cela dépend du serveur web. Les exemples de ce livre utilisent le serveur Apache Tomcat 3.2, l'implémentation de référence de l'API Servlet, écrit entièrement en Java et disponible sous forme de licence « open source » sur le site *http://jakarta.apache.org*. Le serveur Tomcat inclut toute la documentation expliquant le fonctionnement du serveur. C'est pourquoi, même si l'on présente les concepts généraux d'administration du serveur, nous laissons les détails à la documentation du serveur. Si vous choisissez un autre serveur web, ces instructions devraient vous convenir, mais nous ne le garantissons pas.

Si vous utilisez le serveur Apache Tomcat, vous devez mettre le code source de la servlet

dans le répertoire *racine_serveur/webapps/ROOT/WEB-INF/classes* (où *racine_serveur* correspond au répertoire d'installation du serveur). Il s'agit d'un emplacement standard pour les fichiers de classe des servlets. Nous en donnerons les raisons plus loin dans ce chapitre.

Une fois le code source de `HelloWorld` à la bonne place, vous devez le compiler. Le compilateur standard *javac* (ou votre environnement de développement graphique Java favori) peut s'en charger. Soyez simplement certain que les paquetages `javax.servlet` et `javax.servlet.http` sont dans votre classpath. Avec le serveur Tomcat, il suffit d'inclure *racine_serveur/lib/servlet.jar* (ou l'équivalent) quelque part dans votre classpath. Le nom du fichier et son emplacement dépendent du serveur. Reportez-vous à la documentation si vous avez des problèmes. Si vous voyez une page d'erreur indiquant quelque chose comme `Package javax.servlet not found in import`, c'est que les paquetages des servlets n'ont pas été trouvés par le compilateur ; corrigez votre classpath et recommencez.

Maintenant que votre première servlet est compilée, il n'y a plus qu'à démarrer votre serveur et à accéder à la servlet ! Le démarrage du serveur est facile. Recherchez le script *startup.sh* (ou le programme *startup.bat* sous Windows) dans le répertoire *racine_serveur/bin*. Ceci devrait démarrer votre serveur si vous êtes sous Solaris où Windows. Pour d'autres systèmes d'exploitation, vous devrez peut-être modifier un peu les scripts de démarrage. Dans la configuration par défaut, le serveur écoute sur le port 8080.

Il existe plusieurs façons d'accéder à une servlet. Pour cet exemple, nous le ferons explicitement en accédant à une URL où nous aurons ajouté */servlet/* au nom de la classe de la servlet. Vous pouvez entrer cette URL dans votre navigateur favori : *http://*`serveur`*:8080/servlet/HelloWorld*. Remplacez `serveur` par le nom de votre machine serveur ou par `localhost` si le serveur est sur votre machine locale. Vous devriez voir une page similaire à celle montrée en *figure 2-3*.

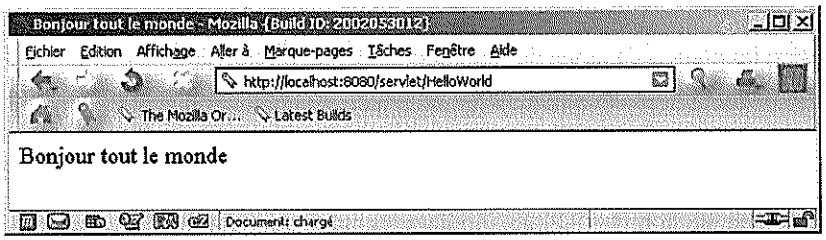


Figure 2-3. La servlet HelloWorld

Si la servlet fait partie d'un paquetage, elle doit être placée dans *racine_serveur/webapps/ROOT/WEB-INF/classes/paquetage* et référencée avec l'URL *http://*`serveur`*:8080/servlet/paquetage.HelloWorld*.

Tous les serveurs n'autorisent pas par défaut l'accès aux servlets à l'aide du préfixe générique */servlet/*. Cette fonctionnalité peut être désactivée pour des raisons de sécurité afin de garantir que les servlets ne sont accédées que *via* des URL spécifiques définies lors de l'administration du serveur. Vérifiez la documentation de votre serveur pour savoir comment activer et désactiver le préfixe */servlet/*.

Gérer les données des formulaires

La servlet « Bonjour tout le monde » n'est pas très attractive, essayons donc de faire quelque chose de plus ambitieux. Cette fois nous allons créer une servlet qui accueille l'utilisateur par son nom. Ce n'est pas difficile. Tout d'abord, nous avons besoin d'un formulaire HTML qui demande à l'utilisateur son nom. La page suivante devrait suffire :

```
<HTML>
<HEAD>
<TITLE>Accueil</TITLE>
</HEAD>
<BODY>
<FORM METHOD=GET ACTION="/servlet/Hello">
Si vous le voulez bien, indiquez-moi votre nom :
<INPUT TYPE=TEXT NAME="nom"><P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

La *figure 2-4* montre comment cette page apparaît à l'utilisateur.

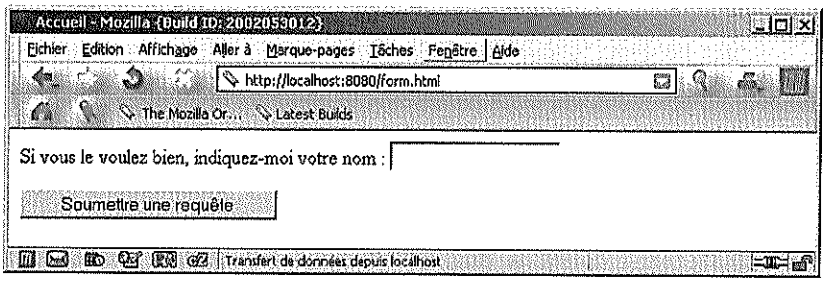


Figure 2-4. Un formulaire HTML

Ce formulaire doit être placé dans un fichier HTML stocké dans le répertoire *racine_document* du serveur. Il s'agit de l'endroit où le serveur recherche les fichiers statiques. Pour le serveur Tomcat, ce répertoire est *racine_serveur/webapps/ROOT*. Si le fichier est placé dans ce répertoire, on y accède *via* l'URL *http://*serveur*:8080/form.html*.

Lorsque l'utilisateur soumet ce formulaire, son nom est envoyé à la servlet `Hello` car nous avons fait pointer l'attribut ACTION sur cette servlet. Le formulaire utilise la méthode GET, les données sont donc ajoutées à l'URL de la requête sous forme d'une chaîne d'interrogation. Par exemple, si l'utilisateur entre le nom « Inigo Montoya », l'URL de la requête est *http://*serveur*:8080/servlet/Hello?nom=Inigo+Montoya*. Le navigateur code de façon spéciale, par un signe +, l'espace qui se trouve dans le nom car les URL ne peuvent pas contenir des espaces.

Un objet `HttpServletRequest` de la servlet lui donne accès aux données du formulaire contenues dans la chaîne d'interrogation. L'*exemple 2-2* montre une version modifiée de notre servlet `Hello` qui utilise un objet requête pour lire le paramètre `nom`.

Exemple 2-2. Servlet sachant à qui elle dit bonjour

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Hello extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter( );

    String nom = req.getParameter("nom");
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Bonjour " + nom + "</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("Bonjour " + nom);
    out.println("</BODY></HTML>");
  }

  public String getServletInfo( ) {
    return "Servlet connaissant le nom de la personne à qui elle " +
           "dit bonjour.";
  }
}
```

Cette servlet est presque identique à la servlet `HelloWorld`. Le principal changement est que maintenant elle appelle `req.getParameter("nom")` pour récupérer le nom de l'utilisateur et qu'elle l'affiche à la place du « tout le monde » plutôt impersonnel. La méthode `getParameter()` donne à une servlet accès aux paramètres de sa chaîne d'interrogation. Elle retourne la valeur du paramètre décodée ou `null` si le paramètre n'a pas été spécifié. Si le paramètre a été envoyé mais sans valeur, comme dans le cas d'un champ de formulaire vide, `getParameter()` retourne une chaîne vide.

Cette servlet ajoute aussi une méthode `getServletInfo()`. Une servlet peut surcharger cette méthode pour retourner une information descriptive sur elle-même, comme son objectif, l'auteur, la version, et/ou un copyright. Cela ressemble à la méthode `getAppletInfo()` des applets. La méthode est principalement utilisée pour placer des informations dans un outil d'administration de serveurs web. Vous noterez que nous ne nous embêterons pas à l'inclure dans les prochains exemples car elle ne ferait que les compliquer.

La sortie de la servlet ressemble à ce qui est présenté à la *figure 2-5*.

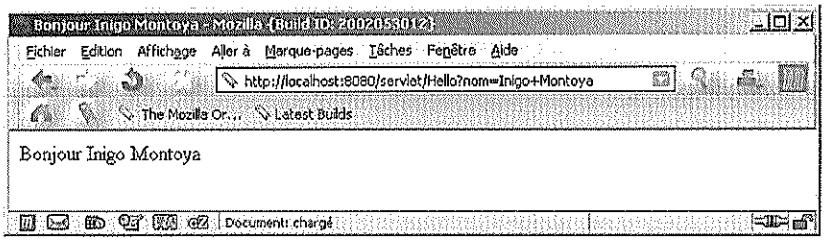


Figure 2-5. Servlet Hello utilisant les données du formulaire

Traiter les requêtes POST

Vous avez maintenant vu deux servlets qui implémentent la méthode `doGet()`. Modifions donc notre servlet `Hello` afin qu'elle gère aussi les requêtes POST. Puisque nous voulons le même comportement avec POST qu'avec GET, nous pouvons simplement aiguiller les requêtes POST vers la méthode `doGet()` à l'aide du code suivant :

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
  doGet(req, res);
}
```

Maintenant, la servlet `Hello` peut gérer les soumissions de formulaire qui utilisent la méthode POST :

```
<FORM METHOD=POST ACTION="/servlet/Hello">
```

En général, il est préférable qu'une servlet implémente, soit `doGet()`, soit `doPost()`. Pour décider laquelle implémenter, il faut savoir quelle forme de requête la servlet va traiter. Le code que vous écrivez pour implémenter ces méthodes est pratiquement le même. La différence principale est que `doPost()` a la possibilité d'accepter de grandes quantités de données en entrée.

Vous pourriez vous demander ce qui serait arrivé à la servlet `Hello` si l'on y avait accédé par une requête POST avant que nous n'implémentions `doPost()`. Le comportement par défaut des méthodes `doGet()` et `doPost()` hérité de `HttpServlet` est de retourner une erreur au client lui indiquant que l'URL demandée ne supporte pas cette méthode.

Traiter les requêtes HEAD

Un petit peu de magie facilite le traitement des requêtes HEAD (envoyées par un client lorsqu'il ne veut voir que les en-têtes de la réponse). Il n'y a aucune méthode `doHead()` à écrire. Toute servlet qui est une sous-classe de `HttpServlet` et qui implémente la méthode `doGet()` supporte automatiquement les requêtes HEAD.

Voici comment cela fonctionne. La méthode `service()` de `HttpServlet` identifie les requêtes HEAD et les traite de façon spéciale. Elle construit un objet `HttpServletResponse` modifié et le passe, avec une requête inchangée, à la méthode `doGet()`. Celle-ci procède comme d'habitude, mais seuls les en-têtes sont créés et retournés au client. C'est l'objet de réponse spécial qui supprime le contenu du corps.

Bien que cette stratégie soit pratique, vous pourrez quelquefois améliorer les performances en détectant les requêtes HEAD dans la méthode `doGet()`, de façon à ce qu'elle

puisse retourner plus rapidement, avant de perdre du temps à écrire des données que personne ne verra. L'*exemple 2-3* utilise la méthode `getMethod()` de la requête pour implémenter cette stratégie (autrement appelé un *hack*) dans notre servlet `Hello`.

Exemple 2-3. Servlet Hello modifiée pour retourner rapidement lors des requêtes HEAD

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Hello extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    // Définir l'en-tête Content-Type.
    res.setContentType("text/html");

    // Retourner plus vite si c'est un HEAD.
    if (req.getMethod( ).equals("HEAD")) return;

    // Continuer sinon.
    PrintWriter out = res.getWriter( );
    String nom = req.getParameter("nom");
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Bonjour " + nom + "</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("Bonjour " + nom);
    out.println("</BODY></HTML>");
  }
}
```

Notez que nous avons défini l'en-tête `Content-Type`, même si nous traitons une requête HEAD. Les en-têtes de ce type sont retournés au client. Certaines valeurs d'en-tête, telles que `Content-Length`, peuvent ne pas être disponibles tant que la réponse n'a pas été calculée. Si vous voulez être rigoureux dans le retour de ces valeurs d'en-tête, l'efficacité de ce raccourci est limitée.

Assurez-vous de terminer le traitement de la requête par une instruction `return`. N'appelez pas `System.exit()`. Si vous le faisiez, vous risqueriez de faire se terminer le serveur web.

Applications web

Une *application web* est un ensemble de servlets, de JSP (*Java-Server Pages*), de documents HTML, d'images, de modèles et d'autres ressources web configurées de manière à être déployées de façon portable sur tout serveur web supportant les servlets. Puisque tout le monde est d'accord sur l'endroit exact où les fichiers d'une application web doivent être stockés et sur un format de fichier de configuration standard, une application web peut être facilement transférée d'un serveur à un autre sans demander une administration supplémentaire du serveur. Ils sont loin les jours où des manuels d'installa-

tion détaillés vous expliquaient comment installer des composants web tiers, avec des instructions différentes pour chaque type de serveur web.

Tous les fichiers du répertoire *racine_serveur/webapps/ROOT* appartiennent à une même application web (l'application racine). Pour simplifier le déploiement, ces fichiers sont réunis dans un même fichier d'archive et ils sont déployés sur un autre serveur en plaçant le fichier d'archive dans un répertoire spécifique. Les fichiers d'archive ont pour extension *.war* (*web application archive*). Les fichiers WAR sont en fait des fichiers JAR (créés avec l'utilitaire *jar*) sauvegardés avec une extension différente. L'utilisation du format JAR permet aux fichiers WAR d'être stockés sous une forme compressée et que leur contenu soit signé numériquement. L'extension *.war* a été préférée à *.jar* afin que les personnes et les outils sachent les traiter différemment.

La structure de fichiers dans une application web est strictement définie. *L'exemple 2-4* montre une liste de fichiers possible.

Exemple 2-4. Structure de fichiers dans une application web

```
index.html
feedback.jsp
images/baniere.gif
images/fleche.gif
WEB-INF/web.xml
WEB-INF/lib/bhawk4j.jar
WEB-INF/classes/MaServlet.class
WEB-INF/classes/com/masociete/interface/CorpServlet.class
WEB-INF/classes/com/masociete/interface/SupportClass.class
```

Cette hiérarchie peut être maintenue comme des fichiers séparés dans un répertoire du serveur ou elle peut être regroupée dans un fichier WAR. À l'installation, cette application web est associée à un préfixe d'URI sur le serveur. L'application web traite ensuite toutes les requêtes commençant par ce préfixe. Par exemple, si la structure de fichiers précédentes est installée sous le préfixe */demo*, le serveur utilise cette application web pour traiter toutes les requêtes commençant par */demo*. La requête de */demo/index.html* renvoie le fichier *index.html* de l'application web. La requête de */demo/feedback.jsp* ou de */demo/images/baniere.gif* retourne aussi le contenu pris dans l'application web.

Le répertoire WEB-INF

Le répertoire *WEB-INF* est spécial. Les fichiers qu'il contient ne sont pas envoyés directement au client ; ils contiennent des classes Java et des informations de configuration pour l'application web. Le répertoire se comporte comme le répertoire *META-INF* d'un fichier JAR : il stocke des méta-informations sur le contenu de l'archive.

Le répertoire *WEB-INF/classes* contient les fichiers de classe des classes support et de classes des servlets de l'application web. *WEB-INF/lib* contient les classes stockées dans des fichiers JAR. Pour des questions pratiques, les chargeurs de classes du serveur examinent automatiquement les répertoires *WEB-INF/classes* et *WEB-INF/lib* quand ils doivent charger des classes — aucune étape d'installation supplémentaire n'est nécessaire.

Les servlets de cette application web sont invoquées à l'aide d'URI tels que */demo/servlet/MaServlet* et */demo/servlet/com.masociete.interface.CorpServlet*. Notez com-

ment chaque requête de cette application web commence par */demo*, même les requêtes pour les servlets.

Avec le serveur Tomcat, *racine_serveur/webapps/ROOT* est le contexte par défaut associé au chemin racine «/». Cela signifie que les servlets placées dans le répertoire *racine_serveur/webapps/ROOT/WEB-INF/classes* sont accessibles, comme nous l'avons vu, *via* le chemin */servlet/HelloWorld*. Avec Tomcat, cette correspondance de contexte par défaut peut être modifiée et de nouvelles correspondances peuvent être ajoutées en modifiant le fichier de configuration du serveur *racine_serveur/conf/server.xml*. D'autres serveurs configurent différemment les correspondances ; consultez la document de votre serveur pour plus de détails.

Le fichier *web.xml* du répertoire *WEB-INF* s'appelle le *descripteur de déploiement*. Ce fichier contient des informations de configuration pour l'application web dans laquelle il réside. Il s'agit d'un fichier XML avec une DTD normalisée. La DTD contient plus de 50 balises, autorisant un contrôle total du comportement de l'application web. Le descripteur de déploiement contrôle l'enregistrement des servlets, la correspondance des URL, les fichiers d'accueil et les types MIME, ainsi que des fonctionnalités avancées comme les contraintes de sécurité au niveau de la page et le comportement des servlets dans un environnement distribué. Nous présenterons le contenu de ce fichier tout au long de ce livre. La DTD commentée est disponible à l'annexe C, *Descripteur de déploiement, référence de la DTD*.

À ce stade, la structure du fichier *web.xml* n'est pas en elle-même importante ; l'important est le fait qu'un descripteur de déploiement permettant d'avoir des informations de configuration données de manière indépendante du serveur simplifie énormément le processus de déploiement. Grâce aux descripteurs de déploiement, non seulement les servlets simples sont portables, mais il est aussi possible de transférer sur d'autres serveurs des sous-parties autonomes de votre site.

Au fil du temps, il est probable qu'un marché commercial va se développer pour les fichiers WAR. Ils vont devenir des composants web enfichables, capables d'être téléchargés, installés et actifs directement, quel que soit le système d'exploitation ou le serveur web.

Les descripteurs de déploiement apportent aussi aux sociétés d'hébergement de sites web un moyen pratique pour supporter plusieurs clients sur le même serveur. Les clients peuvent obtenir un contrôle sur leurs domaines individuels. Ils peuvent gérer eux-mêmes l'enregistrement des servlets, les correspondances d'URL, les types MIME et les contraintes de sécurité sur les pages, sans avoir un accès global au serveur web.

XML et DTD

XML signifie *eXtensible Markup Language*.[a] Il s'agit d'une syntaxe universelle pour la structuration des données, créée comme une activité du World Wide Web Consortium (W3C) début 1996. Depuis sa normalisation en 1998, il a été largement adopté par le web.

XML est similaire à HTML car tous deux prennent du contenu et le marquent à l'aide de balises qui commencent et se terminent par les symboles < et >, par exemple `<titre>` et `</titre>`. Cependant, l'objectif de XML est différent de celui de HTML. Les balises d'un document XML ne définissent pas comment le texte doit être affiché mais elles en donnent la signification. Il s'agit d'un langage de balisage « extensible » car il est possible de créer de nouvelles balises ayant leur propre signification et adaptées au document en cours de rédaction. XML fonctionne très bien avec le format de fichier à plat car c'est une technique standard indépendante de la plate-forme et bien définie permettant de décrire des données hiérarchiques. De plus, il existe de nombreux outils permettant de lire, écrire et manipuler des fichiers XML.

Les règles d'écriture de XML sont plus strictes que celles de HTML. Premièrement, les balises XML sont sensibles à la casse. `<servlet>` et `<SERVLET>` sont deux balises différentes. Deuxièmement, à toute balise d'ouverture doit correspondre une balise de fermeture. S'il existe une balise d'ouverture `<servlet>`, il doit exister une balise de fermeture `</servlet>` — même si pour des raisons pratiques, la syntaxe de balise vide `<servlet/>` peut être substituée à la paire `<servlet></servlet>`. Troisièmement, les éléments imbriqués ne doivent pas se croiser. On peut donc avoir `<dehors><dedans>données</dedans></dehors>`, alors que `<dehors><dedans>data</dehors></dedans>` est invalide. Quatrièmement, et dernièrement, toutes les valeurs d'attributs doivent être entourées de guillemets, simples ou doubles. Cela signifie que `<servlet id="0"/>` est valide, contrairement à `<servlet id=0/>`. Les documents qui suivent ces règles sont dits *bien formés* et ils seront correctement analysés par les outils automatiques.

En dehors de ces règles, il existe des manières de déclarer explicitement une structure des balises d'un fichier XML. Une telle spécification s'appelle une DTD (*Document Type Definition*). Une DTD définit les balises autorisées dans un fichier XML, le type des données contenues par ces balises, ainsi que la hiérarchie dans laquelle ces balises peuvent (ou doivent) être placées. Chaque fichier XML peut être déclaré comme respectant une certaine DTD. Les fichiers qui sont parfaitement conformes à la DTD déclarée sont dits *valides*.

Dans les servlets, XML sert de format de stockage pour les fichiers de configuration. XML peut aussi être utilisé par les servlets pour la création de contenu, comme l'explique le chapitre 17, *XMLC*.

Pour plus d'informations sur XML, visitez le site *http://www.w3.org/XML/* et lisez le livre *Java et XML* de Brett McLaughlin (Éditions O'Reilly).

---

a XML s'appelait avant MAGMA. Voir *http://www.xml.com/axml/notes/TheCorrectTitle.html*.

Le descripteur de déploiement

L'*exemple 2-5* montre un fichier contenant un descripteur de déploiement simple. Pour que ce fichier décrive l'application web par défaut de Tomcat, il doit être placé dans *racine_serveur/webapps/ROOT/WEB-INF/web.xml*.

Exemple 2-5. Un descripteur de déploiement simple

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            hi
        </servlet-name>
        <servlet-class>
            HelloWorld
        </servlet-class>
    </servlet>
</web-app>
```

La première ligne déclare qu'il s'agit d'un fichier XML 1.0 contenant des caractères provenant du jeu de caractères standard ISO-8859-1 (Latin-1). La deuxième ligne indique la DTD du fichier, permettant à un outil de vérifier que le fichier est *valide* et conforme aux règles de la DTD. Tous les fichiers de descripteur de déploiement commencent par ces deux lignes, ou des lignes très similaires.

Le reste, ce qui se trouve entre les balises `<web-app>` et `</web-app>`, donne au serveur des informations sur l'application web. Cet exemple simple enregistre notre servlet `HelloWorld` sous le nom `hi` (les espaces sont supprimés). Le nom d'enregistrement se trouve entre les balises `<servlet-name>` ; le nom de classe est placé entre les balises `<servlet-class>`. La balise `<servlet>` contient les balises `<servlet-name>` et `<servlet-class>`. Il est vrai que la syntaxe XML du descripteur de déploiement semble mieux adaptée à une lecture automatique qu'à une édition directe par une personne. C'est pourquoi, la plupart des fournisseurs de serveurs commerciaux offrent des outils graphiques facilitant la création du fichier *web.xml*. Il existe aussi sur le marché plusieurs éditeurs XML facilitant la création de documents XML.

Après l'enregistrement et après le redémarrage du serveur, nous pouvons accéder à la servlet `HelloWorld` *via* l'URL *http://*`serveur`*:8080/servlet/hi*. Vous vous demandez peut-être pourquoi il faut enregistrer la servlet sous un nom spécial. La réponse courte est que cela permet au serveur de se rappeler des choses sur la servlet et de la traiter de manière spécifique.

Un exemple de traitement spécial est la création de motifs d'URL permettant d'invoquer la servlet enregistrée. Pour le client, l'URL demandée peut ressembler à n'importe quelle autre URL ; cependant, le serveur détecte que cette requête correspond à un mo-

Attention à l'ordre de balises

Sachez que l'ordre des balises dans un fichier *web.xml* est important. Par exemple, la balise `<servlet-name>` doit apparaître avant `<servlet-class>` pour que tout fonctionne. C'est l'ordre indiqué dans la DTD. Les analyseurs syntaxiques validant garantissent cet ordre et déclarent le document invalide si cet ordre n'est pas respecté. Certains serveurs, même ceux qui ne disposent pas d'analyseurs validant, peuvent attendre cet ordre et peuvent être perturbés par un ordre différent. Pour plus de sûreté, assurez-vous que toutes les balises `<web-app>` sont dans le bon ordre. Certaines sont optionnelles, mais chaque balise présente doit être dans le bon ordre. Heureusement, des outils simplifient cette tâche. Consultez la DTD à l'annexe C pour plus d'informations.

tif donné et qu'elle doit donc être traitée par une servlet particulière. Par exemple, nous pouvons décider que l'URL *http://*serveur*:8080/hello.html* invoque la servlet `HelloWorld`. En utilisant les correspondances de servlet de cette manière, on peut cacher le fait que le site exploite les servlets. Il est aussi plus facile de remplacer une page existante sur le serveur afin que tous les signets et les liens vers la page soient toujours valides.

Les motifs d'URL sont configurés dans le descripteur de déploiement, comme le montre l'*exemple 2-6*.

Exemple 2-6. Ajouter une correspondance de servlet

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            hi
        </servlet-name>
        <servlet-class>
            HelloWorld
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>
            hi
        </servlet-name>
        <url-pattern>
            /hello.html
        </url-pattern>
    </servlet-mapping>
</web-app>
```

Ce descripteur de déploiement ajoute une entrée `<servlet-mapping>` pour indiquer au serveur que la servlet nommée `hi` doit prendre en charge toutes les URL correspondant au motif */hello.html*. Si cette application web est associée au chemin racine « / », la servlet `HelloWorld` traite les requêtes *http://*`serveur`*:8080/hello.html*. Si l'application web est associée au préfixe */accueil*, la servlet `Hello` traite alors les requêtes effectuées sur *http://*`serveur`*:8080/accueil/hello.html*.

Il est possible de spécifier différentes règles de correspondance d'URL dans le descripteur de déploiement. Il existe quatre types de correspondance, recherchés dans l'ordre suivant :

- Les correspondances explicites, comme */hello.html* ou */images/graphique.gif*, ne contenant aucun caractère générique. Ce type de correspondance est utile dans les remplacements de pages existantes.
- Les correspondances avec préfixe de chemin, comme */compressé/**, */fichierbd/** ou */catalogue/produits/**. Elles commencent par /, se terminent par /* et gèrent toutes les requêtes commençant par ce préfixe (sans tenir compte du chemin de contexte). Ce type de correspondance permet à une servlet de contrôler une hiérarchie virtuelle complète. Par exemple, la servlet gérant */fichierbd/** peut traiter les fichiers d'une base de données, tandis que la servlet gérant */compressé/** peut servir les fichiers provenant d'un système de fichiers avec compression automatique.
- Les correspondances d'extension, comme **.wm* ou **.jsp*. Elles commencent par * et traitent toutes les requêtes qui se terminent par l'extension. Ces correspondances permettant à une servlet d'opérer sur tous les fichiers possédant une extension donnée. Par exemple, une servlet peut être configurée pour gérer tous les fichiers se terminant par **.jsp* afin de supporter les JavaServer Pages. (En fait, c'est une correspondance implicite imposée par la spécification des servlets.)
- La correspondance par défaut, /. Elle spécifie le serveur par défaut de l'application web, utilisée uniquement lorsqu'aucune autre correspondance n'a été trouvée. Elle est identique à la correspondance avec préfixe de chemin réduit (/*), excepté qu'elle est recherchée après les correspondances d'extension. Cela donne un contrôle sur le traitement des fichiers basiques — une option puissante, mais qu'il ne faut pas utiliser à la légère.

En cas de collision des correspondances, les correspondances exactes sont prioritaires par rapport aux correspondances de préfixe de chemin, qui sont elles-mêmes prioritaires sur les correspondances d'extension. La correspondance par défaut n'est invoquée que si les autres n'ont pas été trouvées. Les plus longues correspondances dans une catégorie sont prioritaires sur les correspondances plus courtes dans une catégorie.

L'extrait d'un descripteur de déploiement à l'*exemple 2-7* montre différentes correspondances permettant d'accéder à la servlet `HelloWorld`.

Exemple 2-7. Tant de manières de dire bonjour

```
<!-- ... -->
<servlet-mapping>
    <servlet-name>
        hi
    </servlet-name>
```

Exemple 2-7. Tant de manières de dire bonjour (suite)

```
        <url-pattern>
            /bonjour.html
        </url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>
            hi
        </servlet-name>
        <url-pattern>
            *.bonjour
        </url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>
            hi
        </servlet-name>
        <url-pattern>
            /bonjour/*
        </url-pattern>
    </servlet-mapping>
    <!-- ... -->
```

Avec ces correspondances, la servlet `HelloWorld` peut être invoquée à l'aide des URL suivantes :

```
/servlet/HelloWorld
/servlet/hi
/bonjour.html
/bienle.bonjour
/rendez/vous/sympa/ici.bonjour
/bonjour/a/vous
```

Nous verrons d'autres correspondances de servlet plus pratiques tout au long de ce livre.

Allez plus loin

Nous sommes bien conscients que ce chapitre a été une introduction très rapide aux servlets, aux applications web et aux fichiers de configuration XML. Nous espérons cependant que vous avez maintenant une idée de l'écriture d'une servlet simple, de son installation sur le serveur et de la configuration des chemins du serveur permettant de l'exécuter. Bien entendu, les servlets peuvent faire bien plus qu'afficher « Bonjour tout le monde » et accueillir les utilisateurs par leur nom. Maintenant que vous avez les pieds mouillés, nous pouvons plonger dans les détails et passer à des applications plus intéressantes.

3

Cycle de vie d'une servlet

Le cycle de vie d'une servlet est l'une des fonctionnalités les plus intéressantes des servlets. Ce cycle de vie est un puissant hybride de ceux utilisés dans la programmation CGI et dans les programmations WAI/NSAPI ou ISAPI, comme on l'a vu au chapitre 1, *Introduction*.

La solution servlet

Le cycle de vie d'une servlet permet aux conteneurs de servlets de répondre aux problèmes de performance et de ressources posés par CGI ainsi qu'aux problèmes de sécurité liés à la programmation des serveurs avec une API de bas niveau. Un conteneur de servlets exécute le plus souvent toutes ses servlets dans une seule machine virtuelle Java (JVM). Étant dans la même JVM, les servlets partagent efficacement des données, tout en respectant les parties privées des autres servlets grâce au langage Java. Les servlets peuvent aussi persister entre les requêtes sous la forme d'instances d'objet. De cette manière, la consommation de la mémoire est bien plus faible qu'avec les processus, même si les servlets peuvent toujours conserver efficacement des références à des ressources externes.

Le cycle de vie d'une servlet est très souple. La seule règle imposée est que le conteneur de servlets doit se conformer au contrat de cycle de vie suivant :

- Créer et initialiser la servlet.
- Traiter les services demandés par les clients, le cas échéant.
- Détruire la servlet et la passer au ramasse-miettes.

Il est parfaitement légal qu'une servlet soit chargée, créée et instanciée dans sa propre JVM, pour être ensuite détruite et passée au ramasse-miettes sans qu'elle n'ait traitée une seule requête cliente ou bien après en n'avoir traité qu'une seule. Cependant, le conteneur de servlets qui ferait de ce fonctionnement une habitude ne resterait probablement pas très longtemps sur le marché. Dans ce chapitre, nous décrivons les implémentations les plus communes du cycle de vie des servlets HTTP.

Une seule machine virtuelle Java

La plupart des conteneurs de servlets exécutent toutes les servlets dans une seule JVM afin d'augmenter les possibilités de partage des informations. (L'exception étant les conteneurs haut de gamme supportant l'exécution de servlets distribuées sur plusieurs serveurs, comme décrit au chapitre 12, *Servlets d'entreprise et J2EE*.) En fonction du serveur, la JVM s'exécute à des endroits différents :

- Avec un serveur écrit en Java, comme le serveur Apache Tomcat, le serveur peut s'exécuter lui-même dans une JVM, au côté de ses servlets.
- Avec un serveur web multithread à processus unique écrit dans un autre langage, la JVM est souvent embarquée dans le processus serveur. De cette manière, on maximise les performances car une servlet devient, en un sens, juste une autre extension de l'API bas niveau du serveur. Un tel serveur peut invoquer une servlet avec un changement léger de contexte et il peut fournir des informations sur les requêtes au travers d'invocations directes de méthodes.
- Un serveur web multiprocessus (qui exécute plusieurs processus pour traiter les requêtes) n'a pas vraiment la possibilité d'embarquer une JVM directement dans son processus car il y en a plusieurs. Ce type de serveur exécute habituellement une JVM externe partagée par ses processus. Avec cette approche, chaque accès à une servlet implique un changement lourd de contexte comme celui de FastCGI. Cependant, toutes les servlets partagent le même processus externe.

Heureusement, du point de vue de la servlet (et donc du nôtre, en tant qu'auteur de servlets), l'implémentation du serveur n'a pas vraiment d'importance car il se comporte toujours de la même façon.

Persistance d'instance

Nous avons dit précédemment que les servlets persistent entre les requêtes sous forme d'instances d'objet. Autrement dit, au moment où le code d'une servlet est chargé, le serveur ne crée qu'une seule instance de classe. Cette instance traite chaque requête effectuée sur la servlet. Cela améliore les performances de trois façons :

- L'empreinte mémoire reste petite.
- Le surcoût lié à la création d'un nouvel objet pour la servlet est éliminé. Une servlet peut être déjà chargée dans la machine virtuelle lorsqu'une requête arrive, requête qui peut alors être traitée immédiatement.
- La persistance est possible. Une servlet peut avoir déjà chargé des choses dont elle risque fort d'avoir besoin lors du traitement de la prochaine requête. Par exemple, une connexion à une base de données peut avoir été ouverte une fois et être réutilisée par la suite. Elle peut même être utilisée par un groupe de servlets. Un autre exemple serait une servlet de gestion de caddies qui charge en mémoire la liste des prix ainsi qu'une information sur les clients qui se sont connectés récemment. Une autre servlet pourrait décider de mettre en cache des pages entières de sortie pour gagner du temps si elle reçoit la même requête.

Non seulement les servlets persistent entre des requêtes, mais c'est aussi le cas des threads créés par les servlets. Ce n'est peut-être pas très utile pour la majorité des servlets, mais cela ouvre d'intéressantes possibilités. Considérons le cas dans lequel un

thread d'arrière-plan effectue certains calculs tandis que d'autres threads affichent les derniers résultats. Cela ressemble fort à une applet d'animation où un thread change l'image et une autre l'affiche à l'écran.

Un compteur simple

Pour expliquer le cycle de vie d'une servlet, nous allons commencer par un exemple simple. L'*exemple 3-1* montre une servlet qui compte et affiche le nombre de fois que l'on y a accédé. Pour des raisons de simplicité, la sortie se fait sous forme de texte. (Rappelez-vous que le code de tous les exemples est disponible en ligne. Voir la *Préface*.)

Exemple 3-1. Un compteur simple

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleCounter extends HttpServlet {

  int count = 0;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    count++;
    out.println("Depuis son chargement, on a accédé à cette servlet " +
                count + " fois.");
  }
}
```

Le code est simple — il ne fait qu'imprimer et incrémenter la variable d'instance appelée `count` — mais il montre la puissance de la persistance. Lorsque le serveur charge cette servlet, il en crée une seule instance pour traiter chaque requête faite sur la servlet. C'est pourquoi le code peut être aussi simple. Les mêmes variables d'instance existent entre les invocations et pour toutes les invocations.

Un compteur synchronisé simple

Du point de vue du développeur de servlets, chaque client est une autre thread qui appelle la servlet *via* les méthodes `service()`, `doGet()`, ou `doPost()`, comme le montre la *figure 3-1*[1].

1. Comprenez-vous comment une seule instance de servlet peut traiter de multiples requêtes en même temps ? Si ce n'est pas le cas, c'est probablement parce que lorsque nous décrivons l'exécution d'un programme nous voyons souvent des instances d'objet effectuer le travail, invoquant les méthodes des autres objets, et ainsi suite. Mais, bien que ce modèle fonctionne pour des cas simples, ce n'est pas comme cela que les choses fonctionnent vraiment. En réalité, tout le vrai travail est réalisé par des threads. Les instances d'objet ne sont rien de plus que des structures de données manipulées par des threads. Ainsi, si deux threads sont en train de s'exécuter, il est tout à fait possible que tous deux utilisent le même objet au même moment.

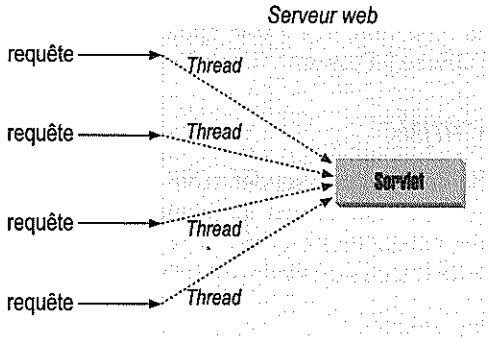


Figure 3-1. Plusieurs threads, une seule instance de la servlet

Si vos servlets ne font que lire à partir de la requête, écrire la réponse et sauver des informations dans les variables locales (c'est-à-dire les variables déclarées à l'intérieur d'une méthode), il n'est pas nécessaire que vous vous occupiez des interactions entre les threads. Cependant, dès qu'une information est sauvée dans des variables non locales (c'est-à-dire des variables déclarées dans la classe mais hors de toute méthode spécifique), vous devez faire attention car chacune de ces threads client a la possibilité de manipuler les variables non locales de la servlet. Si vous ne prenez pas certaines précautions, des données peuvent être corrompues et des incohérences apparaître. Par exemple, la servlet `SimpleCounter` fait indûment la supposition que l'incrémentation du compteur et que l'affichage de sa valeur se font de façon atomique (l'un immédiatement après l'autre, sans interruption). Il est possible que, si deux requêtes sur `SimpleCounter` se font à peu près au même moment, toutes deux afficheront la même valeur pour `count`. Comment ? Imaginez qu'un thread incrémente le compteur et que juste après cela, mais avant que le premier thread n'affiche la valeur, le second thread incrémente aussi le compteur. Chacun affichera alors la même valeur de compteur, après avoir en fait augmenté sa valeur de 2.[2]

L'ordre des exécutions ressemblera à ceci :

```
count++          // Thread 1

count++          // Thread 2

out.println      // Thread 1

out.println      // Thread 2
```

Dans ce cas, l'incohérence n'est pas vraiment un problème, mais beaucoup d'autres servlets ont des opportunités d'erreurs beaucoup plus sérieuses. Pour prévenir ce type de problèmes et les incohérences qui les accompagnent, nous pouvons ajouter un ou plusieurs blocs synchronisés au code. Un bloc ou une méthode synchronisés garantissent que tout le code qu'ils contiennent ne pourra être exécuté de façon concurrente par un

2. Cas bizarre : si `count` était un entier long sur 64 bits au lieu de 32, il serait en théorie possible que l'incrémentation ne soit qu'à moitié exécutée au moment où elle est interrompue par une autre thread. Et cela parce que Java utilise des piles sur 32 bits.

autre thread. Avant qu'un thread ne commence à exécuter un code synchronisé, il doit acquérir un *moniteur* (un *verrou*) sur la classe indiquée. Si un autre thread possède déjà ce moniteur, parce qu'il a déjà exécuté le même bloc synchronisé ou un autre bloc ayant le même moniteur, le premier thread doit attendre. Tout ceci est géré par le langage lui-même, c'est donc très simple à utiliser. Cependant, la synchronisation ne doit être utilisée que lorsque c'est nécessaire. Sur certaines plates-formes, acquérir un moniteur chaque fois que l'on entre dans un bloc synchronisé impose un surcoût non négligeable. Mais ce qui est plus important, tout pendant qu'une thread exécute un bloc synchronisé, les autres threads peuvent être bloqués en attendant que le moniteur soit relâché.

Pour `SimpleCounter`, nous avons quatre options pour traiter ce problème potentiel. Premièrement, nous pouvons ajouter le mot-clé `synchronized` à la signature de `doGet()` :

```
public synchronized void doGet(HttpServletRequest req,
                               HttpServletResponse res)
```

La cohérence est alors garantie par une synchronisation de la méthode entière, en utilisant la classe servlet comme moniteur. En général, ce n'est pas la bonne approche car cela signifie que la servlet ne peut traiter qu'une seule requête GET à la fois.

Notre seconde option est de ne synchroniser que les deux lignes que nous voulons exécuter de façon atomique :

```
PrintWriter out = res.getWriter();
synchronized(this) {
  count++;
  out.println("Depuis son chargement, on a accédé à cette servlet " +
              count + " fois.");
}
```

Cette approche fonctionne beaucoup mieux car elle limite le temps que cette servlet passe dans son bloc synchronisé, tout en accomplissant le même objectif : avoir un compteur cohérent. Bien sûr, pour cet exemple simple, les deux options ne sont pas vraiment différentes.

Notre troisième option est de créer un bloc synchronisé qui effectue tout le travail qui a besoin d'être fait de façon synchronisée, puis d'utiliser le résultat hors du bloc synchronisé. Dans notre exemple, nous pouvons incrémenter le compteur dans un bloc synchronisé, sauver la valeur incrémentée dans une variable locale (une variable déclarée à l'intérieur d'une méthode), puis afficher ensuite la valeur de la variable locale en dehors du bloc synchronisé :

```
PrintWriter out = res.getWriter();
int local_count;
synchronized(this) {
  local_count = ++count;
}
out.println("Depuis son chargement, on a accédé à cette servlet " +
            local_count + " fois.");
```

Ce changement a réduit le bloc synchronisé à sa plus petite taille, tout en maintenant un compteur cohérent.

Notre dernière option est de décider d'ignorer les problèmes de synchronisation et que nous sommes prêts à en supporter les conséquences. Parfois, ces conséquences sont plu-

tôt acceptables. Dans cet exemple, ignorer la synchronisation signifie que certains clients peuvent recevoir un compteur qui n'est pas très exact. En fait, ce n'est pas un gros problème. Cependant, si la servlet était supposée retourner des nombres uniques, ce ne serait pas la même chose.

Bien que ce ne soit pas possible dans cet exemple, une autre option serait de changer les variables d'instance en des variables locales. Les variables locales ne sont pas disponibles aux autres threads, elles n'ont donc pas besoin d'être protégées contre leur corruption. Cependant, les variables locales ne peuvent persister entre deux requêtes, nous ne pouvons donc pas les utiliser pour stocker l'état persistant de notre compteur.

Un compteur général

Ceci posé, le modèle « une instance par servlet » est un peu superficiel. La vérité est que chaque nom enregistré d'une servlet (mais pas chaque correspondance de motif d'URL) est associé à une instance de la servlet. Le nom utilisé pour accéder à la servlet détermine quelle instance traite la requête. Cela a un sens car le client doit avoir l'impression que chaque servlet nommée différemment fonctionne de façon indépendante. Les instances séparées sont aussi une contrainte pour les servlets qui acceptent des paramètres d'initialisation, comme nous en parlerons plus loin dans ce chapitre.

Notre exemple `SimpleCounter` utilise la variable d'instance `count` pour suivre le nombre de fois que l'on y a accédé. Si, à la place, on devait gérer un compteur pour toutes les instances (et donc pour tous les noms enregistrés), on pourrait dans certains cas utiliser une variable de classe, ou variable statique. Ces variables sont partagées par toutes les instances d'une classe. L'*exemple 3-2* montre une servlet qui compte trois choses : le nombre de fois que l'on y a accédé, le nombre d'instances créées par le serveur (une par nom) et le nombre total de fois qu'on les a accédé

Exemple 3-2. Un compteur plus général

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HolisticCounter extends HttpServlet {

  static int classCount = 0;  // Partagée par toutes les instances.
  int count = 0;              // Séparée pour chaque servlet.
  static Hashtable instances = new Hashtable();  // Aussi partagée.

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    count++;
    out.println("Depuis son chargemement, on a accédé à cette servlet " +
                count + " fois.");
```

Exemple 3-2. Un compteur plus général (suite)

```
    // Garder une trace du nombre d'instances en mettant une référence à cette
    // instance dans un Hashtable. Les entrées dupliquées sont ignorées.
    // La méthode size() retourne le nombre d'instances uniques stockées.
    instances.put(this, this);
    out.println("Il y a actuellement " +
                instances.size() + " instances.");

    classCount++;
    out.println("Sur toutes les instances, on a accédé à cette servlet " +
                classCount + " fois.");
  }
}
```

Cet `HolisticCounter` conserve son propre compteur d'accès avec la variable d'instance `count`, un compteur partagé avec la variable de classe `classCount` et le nombre d'instances avec la table de hachage `instances` (une autre ressource partagée qui doit être une variable de classe). Un exemple de sortie est montré en *figure 3-2*.

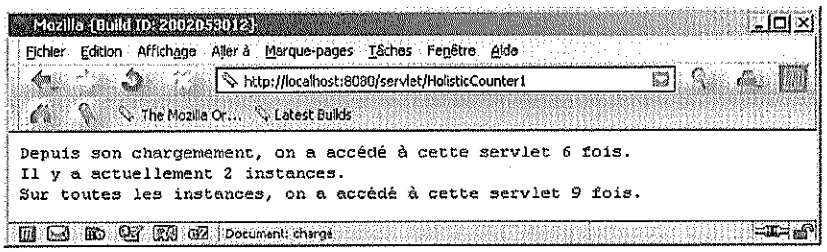


Figure 3-2. Sortie de HolisticCounter

Rechargement d'une servlet

Si vous avez essayé d'utiliser ces compteurs, vous pouvez avoir noté que chaque fois que vous en recompilez un, sa valeur est automatiquement remise à 1. Faites-nous confiance, ce n'est pas un bogue, c'est une fonctionnalité. La plupart des serveurs rechargent automatiquement une servlet dès que leur fichier de classe (dans le répertoire des servlets par défaut, comme *WEB-INF/classes*) change. Une procédure de mise à jour à la volée qui accélère de beaucoup le cycle de développement/test, et qui permet au serveur d'avoir des temps de service plus longs.

On pourrait croire que le rechargement d'une servlet est une fonctionnalité simple, mais c'est en fait une astuce ; presque une bidouille. Les objets `ClassLoader` sont conçus pour ne charger une classe qu'une seule fois. Pour contourner cette limitation et charger les servlets encore et encore, les serveurs utilisent des chargeurs de classes personnalisés qui chargent les servlets à partir du répertoire des servlets par défaut. Cela explique pourquoi les classes de servlets se trouvent dans *WEB-INF/classes*.

Lorsqu'un serveur dirige une requête vers une servlet, il vérifie d'abord si le fichier de classes de la servlet a changé sur le disque. Si c'est le cas, le serveur termine le chargeur de classes utilisé pour charger l'ancienne version et crée une nouvelle instance du char-

geur de classes personnalisé afin de charger la nouvelle version. Pour améliorer les performances, certains serveurs ne vérifient les dates de modification qu'après un certain temps après le contrôle précédent ou sur une demande explicite de l'administrateur.

Dans les versions de l'API Servlet antérieures à la 2.2, cette astuce conduisait au chargement de différentes servlets par différents chargeurs de classes, situation qui provoquait parfois le lancement d'une exception `ClassCastException` quand les servlets partageaient des informations (car une classe chargée par un chargeur de classes n'est pas identique à celle chargée par un second chargeur, même si les données de la classe sous-jacentes sont les mêmes). À partir de l'API Servlet 2.2, il a été établi que ces problèmes de `ClassCastException` ne devaient pas se produire pour les servlets du même contexte. La plupart des implémentations de serveur chargent donc maintenant chaque contexte d'application web au sein d'un même chargeur de classes et utilisent un nouveau chargeur pour recharger le contexte entier quand une servlet du contexte a été modifiée. Puisque toutes les servlets et les classes utilitaires du contexte ont toujours le même chargeur de classes, il n'y aura aucune exception `ClassCastException` inattendue pendant l'exécution. Le rechargement du contexte entier provoque une petite dégradation des performances, mais uniquement pendant le développement.

Le rechargement des classes n'est pas effectué quand seule une classe utilitaire a changé. Pour des raisons d'efficacité, les serveurs ne vérifient que les dates des classes de servlets pour déterminer si le contexte doit être rechargé. Les classes utilitaires présentes dans *WEB-INF/classes* sont rechargées quand le contexte l'est, mais si seule une classe utilitaire a changé, le serveur a peu de chances de le voir.

Le rechargement d'une servlet n'est pas non plus effectué pour les classes (de servlets ou autres) qui se trouvent dans le *classpath* du serveur. Ces classes sont chargées par le chargeur de classes du système, le chargeur de classes principal et non le chargeur personnalisé nécessaire au rechargement. De telles classes sont chargées une seule fois et conservées en mémoire même lorsque leur fichier de classes change.

Il est généralement préférable de mettre les classes utilitaires globales (comme les classes du paquetage `com.oreilly.servlet`) quelque part dans le *classpath* du serveur où elles ne seront pas rechargées. Cela accélère le processus de rechargement et permet aux servlets de contextes différents de partager des instances de ces objets sans générer une exception `ClassCastException`.

Init() et destroy()

Exactement comme les applets, les servlets peuvent définir les méthodes `init()` et `destroy()`. Le serveur appelle la méthode `init()` d'une servlet immédiatement après qu'il a construit l'instance de la servlet et avant qu'elle ne traite une requête. Le serveur appelle la méthode `destroy()` après que la servlet a été retirée et que toutes les requêtes destinées à la servlet ont été traitées ou sont arrivées à échéance.[3]

3. Les premières versions de la spécification de l'API Servlet 2.3 promettent d'ajouter des méthodes pour le cycle de vie permettant aux servlets de savoir quand un contexte ou une session sont créés ou terminés, ainsi que de savoir quand un attribut est lié ou délié d'un contexte ou d'une session.

En fonction du serveur et de la configuration de l'application web, la méthode `init()` peut être appelée à l'un des moments suivants :

- Lorsque le serveur démarre.
- Lors de la première requête à la servlet, juste avant l'invocation de la méthode `service()`.
- Sur demande de l'administrateur du serveur.

Dans tous les cas, il est garanti que `init()` est appelée avant que la servlet ne traite sa première requête.

La méthode `init()` sert typiquement à effectuer l'initialisation de la servlet — créer ou charger des objets qui seront utilisés par la servlet pour le traitement des requêtes. Dans la méthode `init()`, une servlet peut vouloir lire ses paramètres d'initialisation. Ces paramètres sont passés à la servlet elle-même et ne sont pas associés à une quelconque requête. Ils peuvent spécifier des valeurs initiales, comme par exemple à quelle valeur doit commencer un compteur, des valeurs par défaut, ou peut-être un modèle à utiliser si rien n'est spécifié par la requête. Les paramètres d'initialisation d'une servlet se trouvent dans le descripteur de déploiement *web.xml*, même si certains serveurs offrent une interface graphique pour la modification de ce fichier. Voir l'*exemple 3-3*.

Exemple 3-3. Définir les paramètres d'initialisation dans le descripteur de déploiement

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            counter
        </servlet-name>
        <servlet-class>
            InitCounter
        </servlet-class>
        <init-param>
            <param-name>
                initial
            </param-name>
            <param-value>
                1000
            </param-value>
            <description>
                Valeur initiale du compteur  <!-- optionnel -->
            </description>
        </init-param>
    </servlet>
</web-app>
```

Il est possible de mettre plusieurs entrées `<init-param>` dans la balise `<servlet>`. La présence de la balise `<description>` est optionnelle et elle est principalement destinée aux outils graphiques. La DTD complète pour le fichier *web.xml* se trouve à l'annexe F, *Charsets*.

Dans la méthode `destroy()`, une servlet doit libérer toutes les ressources qu'elle a acquises et qui ne pourront être passées au ramasse-miettes. La méthode `destroy()` lui donne aussi une dernière chance d'écrire les informations qui ne sont pas encore sauvegardées et les informations persistantes qui seront lues lors du prochain appel à `init()`.

Un compteur avec init()

Les paramètres d'initialisation peuvent être utilisés pour n'importe quoi. En général, ils spécifient des valeurs initiales ou des valeurs par défaut pour les variables de la servlet, ou bien ils indiquent à la servlet comment personnaliser son comportement. L'*exemple 3-4* étend notre `SimpleCounter` afin qu'il lise un paramètre d'initialisation (nommé `initial`) stockant la valeur initiale du compteur. En donnant à la valeur initiale du compteur une valeur élevée, nous pouvons faire croire que notre page est plus visitée qu'elle ne l'est en réalité.

Exemple 3-4. Compteur lisant des paramètres d'initialisation

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitCounter extends HttpServlet {

  int count;

  public void init() throws ServletException {
    String initial = getInitParameter("initial");
    try {
      count = Integer.parseInt(initial);
    }
    catch (NumberFormatException e) {
      count = 0;
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                          throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    count++;
    out.println("Depuis son chargement (et avec un possible paramètre ");
    out.println("d'initialisation trompeur), on a accédé à cette servlet ");
    out.println(count + " fois.");
  }
}
```

Qu'est devenu super.init(config) ?

Dans l'API Servlet 2.0, une servlet implémentant la méthode `init()` devait implémenter une forme d'`init()` prenant en argument un `ServletConfig` et devait commencer par appeler `super.init(config)` :

```
public void init(ServletConfig config) throws ServletException {
  super.init(config);
  // Code d'initialisation à suivre.
}
```

Le paramètre `ServletConfig` fournissait les informations de configuration à la servlet et l'appel `super.init(config)` passait l'objet de configuration à la classe mère `GenericServlet` dans laquelle il était stocké en vue de son utilisation par la servlet. Plus précisément, la classe `GenericServlet` utilisait le paramètre de configuration pour implémenter l'interface `ServletConfig` (passant tous les appels à la configuration déléguée) afin qu'une servlet puisse appeler les méthodes de `ServletConfig` sur elle-même pour plus de facilité.

L'ensemble du système était plutôt compliqué. Il a été simplifié dans l'API Servlet 2.1 afin qu'une servlet ait juste besoin d'implémenter une version sans argument de `init()` et que la gestion de `ServletConfig` et de `GenericServlet` se charge du reste en coulisses.

La classe `GenericServlet` supporte la méthode `init()` sans argument avec un code similaire au suivant :

```
public class GenericServlet implements Servlet, ServletConfig {

  ServletConfig _config = null;

  public void init(ServletConfig config) throws ServletException {
    _config = config;
    log("init appelée");
    init();
  }

  public void init() throws ServletException { }

  public String getInitParameter(String name) {
    return _config.getInitParameter(name);
  }

  // etc...
}
```

— Suite... —

Notez que le serveur web appelle toujours la méthode `init(ServletConfig config)` de la servlet à l'initialisation. La différence dans l'API 2.1 est que `GenericServlet` passe maintenant cet appel à la méthode `init()` sans argument, que vous pouvez réécrire sans vous occuper de `config`.

Si la compatibilité ascendante est un problème, vous pouvez toujours surcharger `init(ServletConfig config)` et appeler `super.init(config)`. Sinon, vous vous demanderez pourquoi votre méthode `init()` sans argument n'est jamais appelée.

Certains programmeurs trouvent utile de commencer l'implémentation de `destroy()` par un appel à `super.destroy()`. Ceci appelle l'implémentation de `destroy()` fournie par `GenericServlet`, qui écrit dans le journal une note signalant que la servlet a été détruite.

La méthode `init()` utilise la méthode `getInitParameter()` pour obtenir la valeur du paramètre d'initialisation appelé `initial`. Cette méthode prend le nom du paramètre sous la forme d'un `String` et retourne sa valeur sous la forme d'un `String`. Il n'est pas possible d'obtenir cette valeur dans un autre type. La servlet convertit donc la valeur `String` en un entier ou, si un problème survient, prend une valeur par défaut, `0`. Rappelez-vous, si vous testez cet exemple vous devez redémarrer votre serveur que les modifications apportées à *web.xml* soient prises en compte et vous devez faire à nouveau référence à la servlet par son nom enregistré.

Un compteur avec init() et destroy()

Jusqu'à présent, les exemples de compteur ont montré comment l'état d'une servlet persiste entre les accès. Cela ne résout qu'une partie du problème. Chaque fois que le serveur est arrêté ou que la servlet est rechargée, le compteur est ré-initialisé. Ce que nous voulons vraiment est une persistance entre les chargements ; un compteur qui ne repart pas de zéro.

Nous pouvons réaliser cela en utilisant les méthodes `init()` et `destroy()`. L'*exemple 3-5* continue à étendre `InitCounter`, en donnant à la servlet la possibilité de sauvegarder son état dans la méthode `destroy()` et de le recharger dans la méthode `init()`. Pour faire simple, nous supposons que cette servlet n'est pas enregistrée et qu'on y accède uniquement par *http://serveur:port/servlet/InitDestroyCounter*. Si elle était enregistrée sous des noms différents, nous aurions à sauvegarder un état séparé pour chacun des noms.

Exemple 3-5. Un compteur totalement persistant

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitDestroyCounter extends HttpServlet {

  int count;

  public void init() throws ServletException {
```

Exemple 3-5. Un compteur totalement persistant (suite)

```
    // Essayer de charger la valeur initiale à partir de l'état persistant.
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;
    try {
      fileReader = new FileReader("InitDestroyCounter.initial");
      bufferedReader = new BufferedReader(fileReader);
      String initial = bufferedReader.readLine();
      count = Integer.parseInt(initial);
      return;
    }
    catch (FileNotFoundException ignored) { }  // Aucun état sauvé.
    catch (IOException ignored) { }            // Problème à la lecture.
    catch (NumberFormatException ignored) { }  // État sauvé corrompu.
    finally {
      // Fermer le fichier.
      try {
        if (bufferedReader != null) {
          bufferedReader.close();
        }
      }
      catch (IOException ignored) { }
    }

    // Pas de chance avec l'état sauvé, chercher un paramètre d'initialisation.
    String initial = getInitParameter("initial");
    try {
      count = Integer.parseInt(initial);
      return;
    }
    catch (NumberFormatException ignored) { }  // Valeur null ou non entière.

    // On prend par défaut un compteur initial à "0".
    count = 0;
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    count++;
    out.println("Depuis le début, on a accédé à cette servlet " +
                count + " fois.");
  }

  public void destroy() {
    super.destroy();  // Complètement optionnel.
    saveState();
  }
```

Exemple 3-5. Un compteur totalement persistant (suite)

```
  public void saveState() {
    // Essayer de sauvegarder le compteur.
    FileWriter fileWriter = null;
    PrintWriter printWriter = null;
    try {
      fileWriter = new FileWriter("InitDestroyCounter.initial");
      printWriter = new PrintWriter(fileWriter);
      printWriter.println(count);
      return;
    }
    catch (IOException e) {  // Problème à l'écriture.
      // Journaliser l'exception. Voir le chapitre 5.
    }
    finally {
      // Fermer le fichier.
      if (printWriter != null) {
        printWriter.close();
      }
    }
  }
}
```

Chaque fois que cette servlet est déchargée, elle sauvegarde son état dans un fichier appelé *InitDestroyCounter.initial*. Si le chemin n'est pas précisé, le fichier est sauvegardé dans le répertoire courant du processus du serveur, généralement le répertoire de démarrage. Le chapitre 4, *Retrouver l'information*, présente d'autres manières d'indiquer des emplacements différents.[4] Ce fichier contient un seul entier, sauvé sous forme d'une chaîne de caractères, qui représente la dernière valeur du compteur.

Chaque fois que la servlet est chargée, elle essaie de lire la valeur sauvée à partir du fichier. Si, pour quelque raison que ce soit, la lecture échoue (comme c'est le cas la première fois que la servlet s'exécute car le fichier n'existe pas encore), elle vérifie si un paramètre d'initialisation précise la valeur de départ. Si cela échoue aussi, elle démarre à partir de 0. Vous ne serez jamais trop prudent dans les méthodes `init()`.

Les servlets peuvent sauvegarder leur état de différentes façons. Certaines utilisent un format de fichier personnalisé, comme nous l'avons fait ici. D'autres sauvegardent leur état sous forme d'objets Java sérialisés ou le mettent dans une base de données. Il est même possible d'effectuer une *journalisation*, une technique classique dans les bases de données et les sauvegardes sur bandes, dans laquelle l'état complet de la servlet est sauvegardé très peu fréquemment mais un journal stocke les mises à jour incrémentales lorsque quelque chose a changé. La méthode que doit utiliser une servlet dépend de la situation. Dans tous les cas, vous devez toujours faire attention que l'état que vous sauvegardez n'est pas en train d'être changé en arrière-plan.

---

4. L'endroit exact du répertoire courant peut être déterminé en utilisant `System.getProperty("user.dir")`.

Vous vous demandez probablement « que se passe-t-il si le serveur dysfonctionne ? ». C'est une bonne question. La réponse est que la méthode `destroy()` ne sera pas appelée[5]. Ce n'est pas vraiment un problème pour les méthodes `destroy()` qui ne font que libérer des ressources ; un serveur qui redémarre fait très bien ce travail (si ce n'est mieux). Mais c'est un problème pour une servlet qui doit sauvegarder son état dans la méthode `destroy()`. Pour celles-ci, la seule solution garantie est de sauvegarder l'état plus souvent. Une servlet peut décider de sauvegarder son état après chaque traitement d'une requête, comme une servlet « serveur de jeu d'échec » devrait le faire de façon à ce que, même si le serveur redémarre, le jeu puisse reprendre avec la dernière configuration de l'échiquier. D'autres servlets peuvent avoir besoin de sauvegarder leur état seulement lorsque des changements importants sont intervenus ; une servlet « gestion de caddie » a besoin de sauver son état seulement lorsque le client a ajouté ou enlevé un produit de son caddie. Pour d'autres servlets, le fait de perdre un peu des derniers changements de l'état ne pose pas de problème. Ces servlets peuvent sauvegarder leur état après un certain nombre de requêtes. Par exemple, dans `InitDestroyCounter`, il est tout à fait satisfaisant de sauvegarder l'état tous les dix accès. Pour implémenter cela, nous pouvons ajouter la ligne suivante à la fin de `doGet()` :

```
if (count % 10 == 0) saveState();
```

Attention ! Pensez aux problèmes de synchronisation. Nous venons de créer une possibilité de perte de données si `saveState()` est exécutée par deux threads en même temps et la possibilité que `saveState()` ne soit pas appelée du tout si `count` est incrémenté par plusieurs threads à la suite avant le test. Notez que cette possibilité n'existait pas lorsque `saveState()` était appelée uniquement dans la méthode `destroy()` : la méthode `destroy()` est appelée une seule fois par instance de servlet. Maintenant que `saveState()` est appelée dans la méthode `doGet()`, nous devons reconsidérer le problème. Si l'on accède à cette servlet si fréquemment qu'il existe plus de dix threads concurrents, il y a de grandes chances que deux servlets seront dans `saveState()` au même moment. Nous pourrions obtenir un fichier de données corrompu. Il est aussi possible que deux threads incrémentent `count` avant que l'un ou l'autre ne s'aperçoive qu'il est temps d'appeler `saveState()`. Il est facile de résoudre ce problème. Il suffit de déplacer le test du compteur dans un bloc synchronisé où l'on incrémente `count` :

```
int local_count;
synchronized(this) {
  local_count = ++count;
  if (count % 10 == 0) saveState();
}
out.println("Depuis son chargement, on a accédé à cette servlet " +
            local_count + " fois.");
```

La morale de cette histoire est sans pitié : soyez toujours vigilant et protégez le code de votre servlet des problèmes d'accès multithread.

---

5. Si vous n'avez vraiment pas de chance, il est possible que votre serveur se plante dans la méthode `destroy()`. Dans ce cas, vous pourriez avoir un fichier d'état partiellement écrit et donc avoir de fausses données à la place de l'état précédent. Pour être complètement fiable, une servlet devrait sauver son état dans fichier temporaire et ensuite copier ce fichier à la place du fichier d'état officiel en une seule commande.

Modèle mono-thread

Bien qu'en général on ait une instance de servlet par nom de servlet enregistré, il est possible qu'une servlet choisisse d'avoir une réserve d'instances pour chacun de ses noms, toutes ayant à charge de traiter les requêtes. De telles servlets indiquent ce souhait en implémentant l'interface `javax.servlet.SingleThreadModel`. C'est une interface vide ne définissant ni méthode ni variable et servant seulement à marquer la servlet comme souhaitant un cycle de vie différent.

Un serveur qui charge une servlet `SingleThreadModel` doit garantir, conformément à la documentation de l'API Servlet, « que jamais deux threads n'exécuteront concurremment la méthode de service de cette servlet ». Pour accomplir cela, chaque thread utilise une instance de servlet libre prise dans la réserve, comme montré à la *figure 3-3*. Ainsi, toute servlet implémentant `SingleThreadModel` peut être considérée comme sûre vis-à-vis des threads et elle n'impose donc pas que les accès à ses variables d'instance soient synchronisés. Certains serveurs autorisent la configuration du nombre d'instances dans la réserve, d'autres non. Certains serveurs utilisent des réserves avec une seule instance, conduisant au même comportement qu'avec une méthode `service()` synchronisée.

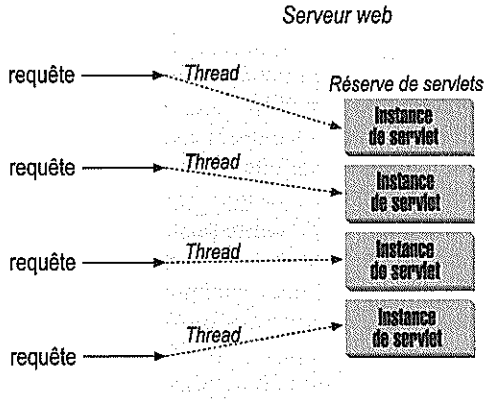


Figure 3-3. Le modèle mono-thread

Un tel cycle de vie `SingleThreadModel` est sans intérêt pour un compteur ou une autre servlet qui demande une gestion centralisée de son état. En revanche, il peut être utile pour éviter une synchronisation lorsque l'on veut un traitement efficace des requêtes.

Par exemple, une servlet qui se connecte à une base de données a parfois besoin d'effectuer de façon atomique des commandes sur la base. Chaque transaction avec la base de données demande un objet de connexion dédié afin que la servlet puisse en quelque sorte garantir qu'il n'existe pas deux threads accédant à la même connexion au même moment. cela peut se faire en utilisant la synchronisation, laissant la servlet gérer une seule requête à la fois. En implémentant `SingleThreadModel` et en ayant une variable de « connexion » par instance de servlet, une servlet peut facilement traiter des requêtes concurrentes car chaque instance a sa propre connexion. Le squelette d'un tel code est montré à l'*exemple 3-6*.

Exemple 3-6. Gérer les connexions à une base de données avec SingleThreadModel

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SingleThreadConnection extends HttpServlet
                                    implements SingleThreadModel {

  Connection con = null;  // Connexion à la base de données, une par instance de
                          //  servlet de la réserve.
  public void init() throws ServletException {
    // Établir la connexion pour cette instance.
    try {
      con = establishConnection();
      con.setAutoCommit(false);
    } catch (SQLException e) {
      throw new ServletException(e.getMessage());
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    try {
      // Utiliser la connexion affectée uniquement à cette instance.
      Statement stmt = con.createStatement();

      // Mettre à jour la base de données.

      // Valider la transaction.
      con.commit();
    } catch (SQLException e) {
      try { con.rollback(); } catch (SQLException ignored) { }
    }
  }

  public void destroy() {
    if (con != null) {
      try { con.close(); } catch (SQLException ignored) { }
    }
  }

  private Connection establishConnection() throws SQLException {
    // Non implémentée. Voir le chapitre 9.
  }
}
```

En réalité, SingleThreadModel n'est pas le meilleur choix pour ce type d'application. Une bien meilleure approche consiste à laisser la servlet utiliser un objet ConnectionPool dédié, placé dans une variable de classe ou d'instance, duquel elle peut retirer ou replacer des connexions. La connexion retirée peut être stockée dans une variable locale afin d'assurer un accès dédié. Une réserve externe donne à la servlet plus de contrôle du la gestion des connexions. La réserve peut aussi vérifier si chaque connexion est active et elle peut être configurée pour toujours créer un nombre minimum de connexions sans jamais dépasser un nombre maximum. Avec SingleThreadModel, le serveur peut créer bien plus d'instances (et donc plus de connexions) que la base de données ne peut accepter.

L'opinion communément admise aujourd'hui est d'éviter l'emploi de SingleThreadModel. La plupart des servlets utilisant SingleThreadModel peuvent être mieux implémentées à l'aide de la synchronisation et/ou des réserves de ressources. Il est vrai que l'interface fournit un certain confort aux programmeurs non familiers de la programmation multithread ; cependant, bien que SingleThreadModel rende la servlet sûre vis-à-vis des threads, ce n'est pas le cas pour le système. L'interface n'empêche pas les problèmes de synchronisation provenant de servlets accédant à des ressources partagées comme des variables statiques ou des objets hors de la portée de la servlet. Il y aura toujours des problèmes avec les threads dans un système multithread, avec ou sans SingleThreadModel.

Traitement en arrière-plan

Les servlets peuvent faire plus que se contenter de persister entre des accès. Elles peuvent aussi travailler. Tout thread démarré par la servlet peut continuer à effectuer des tâches même après que la réponse a été envoyée. Cette possibilité s'avère surtout utile pour des tâches longues dans lesquelles des résultats incrémentaux doivent être rendus disponibles à de multiples clients. Un thread d'arrière-plan démarré dans init() continue à travailler pendant que les threads de traitement des requêtes affichent l'état courant avec doGet(). C'est une technique similaire à celle utilisée dans les applets d'animation, où un thread change l'image pendant qu'un autre l'affiche.

L'*exemple 3-7* montre une servlet qui recherche des nombres premiers à partir d'un quadrillion. Elle démarre avec ce nombre très grand afin que le calcul soit suffisamment lent pour démontrer les effets de cache — quelque chose dont nous aurons besoin dans la prochaine section. L'algorithme utilisé ne pourrait être plus simple : il choisit des nombres impairs candidats et tente de les diviser par chaque entier impair compris entre 3 et leur racine carrée. Si aucun entier n'est un diviseur du nombre candidat, celui-ci est déclaré premier.[6]

6. Pourquoi recherchons-nous uniquement les facteurs inférieurs à la racine carrée ? Simplement parce que tout facteur supérieur à la racine carrée doit correspondre à un facteur inférieur à la racine carrée. S'il n'y a pas de facteur avant la racine carrée, nous savons qu'il n'y en a pas après.

Exemple 3-7. À la chasse aux nombres premiers

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PrimeSearcher extends HttpServlet implements Runnable {

  long lastprime = 0;                    // Dernier nombre premier trouvé.
  Date lastprimeModified = new Date();   // Quand il a été trouvé.
  Thread searcher;                       // Thread de recherche en arrière-plan.

  public void init() throws ServletException {
    searcher = new Thread(this);
    searcher.setPriority(Thread.MIN_PRIORITY);  // Être un bon citoyen.
    searcher.start();
  }

  public void run() {
    //              QTTTBBBMMMTTTOOO
    long candidate = 1000000000000001L;  // Un quadrillion plus 1.

    // Commencer la boucle de recherche des nombres premiers.
    while (true) {                       // Chercher indéfiniment.
      if (isPrime(candidate)) {
        lastprime = candidate;           // Nouveau nombre premier.
        lastprimeModified = new Date();  // Trouvé maintenant.
      }
      candidate += 2;                    // Un nombre pair n'est pas premier.

      // Entre deux candidats faire une pause de 0,2 seconde.
      // Une autre façon d'être un bon citoyen envers les ressources du système.
      try {
        searcher.sleep(200);
      }
      catch (InterruptedException ignored) { }
    }
  }

  private static boolean isPrime(long candidate) {
    // Diviser le nombre par tous les nombre impairs entre 3 et sa √.
    long sqrt = (long) Math.sqrt(candidate);
    for (long i = 3; i <= sqrt; i += 2) {
      if (candidate % i == 0) return false;  // On a trouvé un facteur.
    }

    // Il n'était même pas divisible, c'est donc un nombre premier.
    return true;
  }
```

Exemple 3-7. À la chasse aux nombres premiers (suite)

```
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    if (lastprime == 0) {
      out.println("Toujours à la recherche d'un nombre premier...");
    }
    else {
      out.println("Le dernier nombre premier trouvé était " + lastprime);
      out.println(" à " + lastprimeModified);
    }
  }

  public void destroy() {
    searcher.stop();
  }
}
```

Le thread de recherche commence son travail dans la méthode `init()`. Sa dernière découverte est sauvée dans `lastprime`, et l'heure à laquelle elle a été faite dans `lastprimeModified`. Chaque fois qu'un client accède à la servlet, la méthode `doGet()` renvoie le dernier nombre premier trouvé et l'heure à laquelle cela s'est produit. Le thread de recherche travaille indépendamment des accès client ; même si personne n'accède à la servlet, il continue à rechercher des nombres premiers. Si plusieurs clients accèdent à la servlet au même moment, ils voient tous le même état courant.

Notez que la méthode `destroy()` arrête le thread de recherche.[7] C'est très important ! Si une servlet ne stoppe pas ses threads d'arrière-plan, ces derniers continuent à travailler jusqu'à l'arrêt de la machine virtuelle. Même lorsqu'une servlet est rechargée (soit explicitement, soit parce que son fichier de classes a changé), ses threads ne seront pas stoppés. Au contraire, il y a de grandes chances que la nouvelle servlet crée de nouvelles copies des threads d'arrière-plan.

Chargement au démarrage

Pour que `PrimeSearcher` démarre sa recherche de nombres premiers aussi vite que possible, nous pouvons configurer l'application web de la servlet de manière à ce qu'elle charge la servlet au lancement du serveur. Cela se fait en ajoutant la balise `<load-on-startup>` à l'entrée `<servlet>` dans le descripteur de déploiement, comme le montre l'*exemple 3-8*.

7. L'arrêt des threads en utilisant la méthode `stop()`, comme montré ici, est désapprouvé dans le JDK 1.2 et remplacé par un système plus sûr basé sur des drapeaux. Un thread doit périodiquement examiner un « drapeau » pour déterminer quand elle doit s'arrêter. À ce moment-là, il peut s'arrêter proprement et retourner de sa méthode `run()`. Reportez-vous à la documentation du JDK pour plus de détails.

Exemple 3-8. Charger une servlet au démarrage

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            ps
        </servlet-name>
        <servlet-class>
            PrimeSearcher
        </servlet-class>
        <load-on-startup/>
    </servlet>
</web-app>
```

Cela indique au serveur de créer une instance de `PrimeSearcher` sous le nom enregistré `ps` et d'appeler la méthode `init()` de la servlet pendant la phase de démarrage du serveur. Il est ensuite possible d'accéder à la servlet *via* l'URL */servlet/ps*. Notez que l'instance de la servlet traitant l'URL */servlet/PrimeSearcher* n'est pas chargée au démarrage.

La balise `<load-on-startup>` montrée à l'*exemple 3-8* est vide. Elle peut aussi contenir un nombre positif indiquant l'ordre dans lequel la servlet doit être chargée par rapport aux autres servlets du contexte. Les servlets ayant les nombres les plus bas sont chargées en premières. Les servlets ayant des valeurs négatives peuvent être chargées à n'importe quel moment dans la séquence de démarrage, l'ordre exact dépendant du serveur. Par exemple, le fichier *web.xml* montré à l'*exemple 3-9* garantit que `first` est chargé avant second, alors que `anytime` peut être chargé à tout moment pendant le démarrage du serveur.

Exemple 3-9. Un petit défilé de servlets

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            first
        </servlet-name>
        <servlet-class>
            First
        </servlet-class>
        <load-on-startup>10</load-on-startup>
```

Exemple 3-9. Un petit défilé de servlets (suite)

```
    </servlet>
    <servlet>
        <servlet-name>
            second
        </servlet-name>
        <servlet-class>
            Second
        </servlet-class>
        <load-on-startup>20</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>
            anytime
        </servlet-name>
        <servlet-class>
            Anytime
        </servlet-class>
        <load-on-startup/>
    </servlet>
</web-app>
```

Mise en cache côté client

À présent, nous sommes persuadés que vous savez que les servlets traitent les requêtes GET avec la méthode `doGet()`. Et c'est presque vrai. La vérité totale est que chaque requête n'a pas forcément besoin d'invoquer `doGet()`. Par exemple, un navigateur web qui accède de façon répétée à `PrimeSearcher` n'a besoin d'appeler `doGet()` que si le thread de recherche a trouvé un nouveau nombre premier. Jusqu'à ce moment, tout appel à `doGet()` génère simplement la même page que l'utilisateur a déjà vue, une page probablement stockée dans le cache du navigateur. En fait, il faut que la servlet puisse indiquer que sa sortie a changé. C'est là qu'intervient la méthode `getLastModified()`.

La plupart des serveurs web, quand ils renvoient un document, incluent dans leur réponse un en-tête `Last-Modified`. Voici un exemple de valeur pour l'en-tête `Last-Modified` :

```
Tue, 06-May-98 15:41:02 GMT
```

Cet en-tête indique au client le moment auquel la page a été modifiée. Seule, cette information n'est que partiellement intéressante, mais elle montre toute son utilité quand un navigateur recharge une page.

La plupart des navigateurs web, quand ils rechargent une page, incluent dans leur requête un en-tête `If-Modified-Since`. Sa structure est identique à l'en-tête `Last-Modified` :

```
Tue, 06-May-98 15:41:02 GMT
```

Cet en-tête indique au serveur la date de dernière modification de la page que le navigateur a téléchargée. Le serveur peut lire cet en-tête et déterminer si le fichier a été modifié depuis cette date. Si c'est le cas, le serveur doit envoyer le nouveau contenu. Si ce

n'est pas le cas, le serveur peut répondre par une réponse simple et courte indiquant au navigateur que la page n'a pas changé. Il suffit alors au navigateur de réafficher la version du document qui se trouve dans son cache. Pour ceux qui sont familiers d'HTTP, cette réponse est le code d'état 304 Not Modified.

Cette technique fonctionne bien pour les pages statiques : le serveur utilise le système de fichiers pour déterminer quand un fichier a été modifié pour la dernière fois. En revanche, pour le contenu généré dynamiquement, comme celui retourné par les servlets, le serveur a besoin d'un peu d'aide. En lui-même, le serveur peut au mieux jouer la sécurité et supposer que le contenu change à chaque accès, éliminant l'utilité des en-têtes `Last-Modified` et `If-Modified-Since`.

Une servlet peut apporter une aide supplémentaire en implémentant la méthode `getLastModified()`. Elle doit implémenter cette méthode pour retourner la date de dernière modification de sa sortie. Un serveur appelle cette méthode deux fois. La première fois, avant de retourner sa réponse afin de déterminer la valeur de l'en-tête `Last-Modified`. La seconde fois, dans le traitement des requêtes GET qui incluent l'en-tête `If-Modified-Since` (généralement un rechargement) pour que le serveur puisse déterminer sa réponse intelligemment. Si la date retournée par `getLastModified()` est égale ou antérieure à la date envoyée dans l'en-tête `If-Modified-Since`, le serveur envoie le code d'état Not Modified. Sinon, il appelle `doGet()` et renvoie la sortie de la servlet.[8]

Certaines servlets peuvent avoir des difficultés pour déterminer la date de dernière modification. Dans ce cas, il est préférable de jouer la sécurité. Cependant, beaucoup de servlets devraient avoir aucun, ou peu de, problèmes. Considérons une servlet implémentant un panneau d'affichage sur lequel les gens postent des offres de co-voiturage ou des demandes de partenaires au tennis. Elle peut enregistrer et renvoyer la date de dernière modification du contenu du panneau d'affichage. Même si la même servlet gère plusieurs panneaux, elle peut retourner une date de modification différente en fonction du panneau indiqué dans les paramètres de la requête. Voici une méthode `getLastModified()` pour notre exemple `PrimeSearcher` qui retourne la date à laquelle le dernier nombre premier a été trouvé :

```
public long getLastModified(HttpServletRequest req) {
  return lastprimeModified.getTime() / 1000 * 1000;
}
```

Notez que cette méthode renvoie une valeur de type `long` représentant la date en nombre de millisecondes écoulées depuis le 1er janvier 1970, GMT. Il s'agit de la même représentation que celle utilisée en interne par Java pour stocker les dates. Ainsi, la servlet utilise la méthode `getTime()` pour retrouver `lastprimeModified` comme un `long`.

Avant de renvoyer cette valeur, la servlet doit l'arrondir à la seconde inférieure la plus proche en la divisant puis en la multipliant par 1 000. Toutes les dates retournées par `getLastModified()` doivent être arrondies de cette manière. En effet, les en-têtes `Last-Modified` et `If-Modified-Since` doivent correspondre à la seconde la plus proche. Si `getLastModified()` retourne la même date mais avec une précision plus élevée, elle

8. Une servlet peut directement fixer son en-tête `Last-Modified` dans `doGet()` en employant des techniques présentées au chapitre 5, *Envoyer une information HTML*. Cependant, au moment où l'en-tête est défini dans `doGet()`, il est déjà trop tard pour décider d'appeler ou non `doGet()`.

peut paraître, de façon erronée, postérieure de quelques millisecondes à celle donnée par `If-Modified-Since`. Par exemple, supposons que `PrimeSearcher` trouve un nombre premier a exactement 869 127 442 359 millisecondes depuis le début de la décennie du disco. Ce fait est indiqué au navigateur, mais seulement à la seconde près :

```
Thu, 17-Jul-97 09:17:22 GMT
```

Maintenant, supposons que l'utilisateur recharge la page et que le navigateur indique au serveur, *via* l'en-tête `If-Modified-Since`, la date à laquelle il pense que la page dans le cache a été modifiée :

```
Thu, 17-Jul-97 09:17:22 GMT
```

Certains serveurs reçoivent cette date, la convertissent en exactement 869 127 442 000 millisecondes, déterminent qu'elle est antérieure de 359 millisecondes à la date retournée par `getLastModified()` et supposent à tort que le contenu de la servlet a changé.[9] C'est pourquoi, pour jouer la sécurité, `getLastModified()` doit toujours arrondir au millième de secondes précédent la date qu'elle calcule.

L'objet `HttpServletRequest` est passé à `getLastModified()` au cas où la servlet a besoin de baser ses résultats sur des informations spécifiques à une requête particulière. Par exemple, la servlet générique pour le panneau d'affichage peut exploiter ces informations pour déterminer quel panneau est demandé.

Mise en cache côté serveur

La méthode `getLastModified()` peut être utilisée, avec un peu d'astuce, pour faciliter la gestion d'un cache côté serveur contenant la sortie des servlets. Les servlets implémentant cette astuce voient leur sortie interceptée et placée dans un cache sur le serveur. Ce contenu est ensuite renvoyé au client en accord avec la méthode `getLastModified()` de la servlet. Cela permet d'améliorer de façon importante la génération des pages par les servlets, en particulier pour celles dont la production de sortie prend beaucoup de temps mais dont le contenu change rarement, par exemple les servlets affichant les résultats d'une base de données.

Pour bénéficier de ce cache côté serveur, une servlet doit :

- étendre `com.oreilly.servlet.CacheHttpServlet` au lieu de `HttpServlet` ;
- implémenter une méthode `getLastModified(HttpServletRequest)` habituelle.

L'*exemple 3-10* montre une servlet tirant parti de `CacheHttpServlet`. Il s'agit d'un livre d'or affichant les commentaires envoyés par les utilisateurs. La servlet stocke ces commentaires en mémoire dans un `Vector` d'objets `GuestbookEntry`. Nous verrons au chapitre 9, *Se connecter aux bases de données*, l'exécution d'une version de cette servlet exploitant une base de données. Pour le moment, afin de simuler la lecture à partir d'une base de données lente, l'affichage boucle pendant une demi-seconde pour chaque entrée. Plus la liste devient longue, plus le rendu de la page est lent. Cependant, puisque la servlet étend `CacheHttpServlet`, le rendu n'est effectué que lors de la première requê-

9. Ce bogue a existé à un moment donné dans la version officielle de la classe `javax.servlet.http.HttpServlet`. Jason a eu le privilège de le corriger quand le code est devenu disponible en *open source*.

te GET suivant l'ajout d'un nouveau commentaire. Toutes les requêtes GET suivantes envoient la réponse qui se trouve dans le cache. Un exemple de sortie est montrée à la *figure 3-4*.

Exemple 3-10. Un livre d'or utilisant CacheHttpServlet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.CacheHttpServlet;

public class Guestbook extends CacheHttpServlet {

  private Vector entries = new Vector();  // Liste des utilisateurs.
  private long lastModified = 0;       // Date d'ajout de la dernière entrée.

  // Afficher les entrées courantes, puis demander une nouvelle entrée.
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    printHeader(out);
    printForm(out);
    printMessages(out);
    printFooter(out);
  }

  // Ajouter une nouvelle entrée, puis revenir à doGet().
  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    handleForm(req, res);
    doGet(req, res);
  }

  private void printHeader(PrintWriter out) throws ServletException {
    out.println("<HTML><HEAD><TITLE>Guestbook</TITLE></HEAD>");
    out.println("<BODY>");
  }

  private void printForm(PrintWriter out) throws ServletException {
    out.println("<FORM METHOD=POST>");  // Se l'envoyer.
    out.println("<B>Veuillez envoyer votre commentaire :</B><BR>");
    out.println("Votre nom : <INPUT TYPE=TEXT NAME=name><BR>");
    out.println("Votre e-mail : <INPUT TYPE=TEXT NAME=email><BR>");
    out.println("Commentaire : <INPUT TYPE=TEXT SIZE=50 NAME=comment><BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\"Envoyer\"><BR>");
    out.println("</FORM>");
```

Exemple 3-10. Un livre d'or utilisant CacheHttpServlet (suite)

```
    out.println("<HR>");
  }

  private void printMessages(PrintWriter out) throws ServletException {
    String name, email, comment;

    Enumeration e = entries.elements();
    while (e.hasMoreElements()) {
      GuestbookEntry entry = (GuestbookEntry) e.nextElement();
      name = entry.name;
      if (name == null) name = "Utilisateur inconnu";
      email = entry.email;
      if (name == null) email = "E-mail inconnu";
      comment = entry.comment;
      if (comment == null) comment = "Aucun commentaire";
      out.println("<DL>");
      out.println("<DT><B>" + name + "</B> (" + email + ") a dit");
      out.println("<DD><PRE>" + comment + "</PRE>");
      out.println("</DL>");

      // S'endormir une demi-seconde pour simuler une source de données lente.
      try { Thread.sleep(500); } catch (InterruptedException ignored) { }
    }
  }

  private void printFooter(PrintWriter out) throws ServletException {
    out.println("</BODY>");
    out.println("</HTML>");
  }

  private void handleForm(HttpServletRequest req,
                          HttpServletResponse res) {
    GuestbookEntry entry = new GuestbookEntry();

    entry.name = req.getParameter("name");
    entry.email = req.getParameter("email");
    entry.comment = req.getParameter("comment");

    entries.addElement(entry);

    // Signaler que nous avons une nouvelle date de dernière modification.
    lastModified = System.currentTimeMillis();
  }

  public long getLastModified(HttpServletRequest req) {
    return lastModified;
  }
}
```

Exemple 3-10. Un livre d'or utilisant CacheHttpServlet (suite)

```
class GuestbookEntry {
  public String name;
  public String email;
  public String comment;
}
```

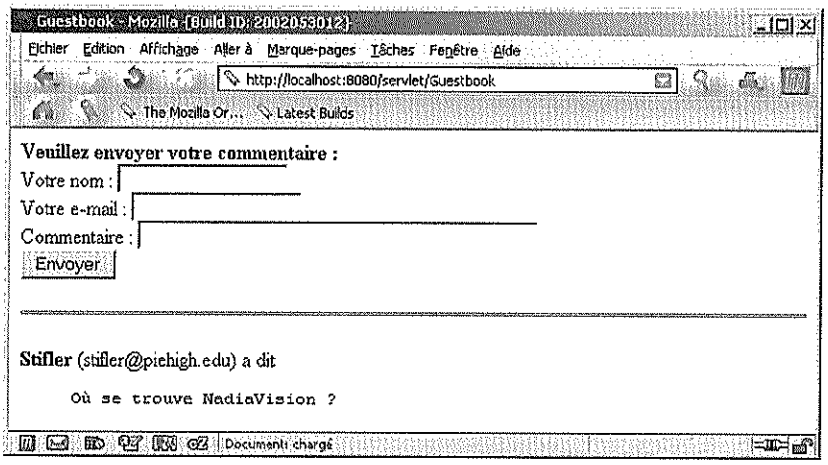


Figure 3-4. Affichage du livre d'or

Le source de `CacheHttpServlet` est montré dans l'*exemple 3-11*. Le code est plutôt complexe, en particulier à ce stade du livre. Si cela vous intéresse, vous pouvez lire le code de cette classe après avoir lu le chapitre 5.

Avant de traiter une requête, la classe `CacheHttpServlet` vérifie la valeur de `getLastModified()`. Si la sortie dans le cache est au moins aussi récente que la date de dernière modification de la servlet, elle est envoyée sans appeler la méthode `doGet()` de la servlet.

Pour jouer la sécurité, si cette classe détecte que la chaîne d'interrogation de la servlet, l'information de chemin supplémentaire ou le chemin de la servlet a changé, le cache est invalidé et recréé. Cependant, cette classe n'invalide pas le cache d'après les en-têtes de requête ou les cookies ; pour les servlets qui modifient leur sortie en fonction de ces informations (par exemple, une servlet de suivi de session), cette classe ne doit pas être utilisée, ou la méthode `getLastModified()` doit prendre en compte les en-têtes et les cookies. Les requêtes POST n'utilisent pas le cache.

`CacheHttpServletResponse` et `CacheServletOutputStream` sont des classes utilitaires et elles ne doivent pas être utilisées directement. La classe a été construite sur l'API Servlet 2.2. Elle fonctionne parfaitement avec une version précédente de l'API Servlet ; elle ne fonctionnera certainement pas avec les futures versions de l'API car l'interface `HttpServletResponse` que `CacheHttpServletResponse` doit implémenter sera probablement modifiée et l'implémentation de certaines de ses méthodes sera vide. Si vous rencontrez un tel problème, une version courante de cette classe est toujours disponible à *http://www.servlets.com*.

Il est intéressant de voir comment CacheHttpServlet intercepte la requête pour son pré-traitement. Elle implémente la méthode service(HttpServletRequest, HttpServletResponse) que le serveur appelle pour passer le contrôle de la gestion de la requête à la servlet. L'implémentation standard de cette méthode par HttpServlet dirige la requête vers doGet(), doPost() ou les autres méthodes selon la méthode HTTP de la requête. CacheHttpServlet surcharge cette implémentation et obtient donc le contrôle du traitement de la requête. Quand la classe a terminé son traitement, elle peut repasser le contrôle à l'implémentation de HttpServlet par un appel à super.service().

Exemple 3-11. La classe CacheHttpServlet

```
package com.oreilly.servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public abstract class CacheHttpServlet extends HttpServlet {

  CacheHttpServletResponse cacheResponse;
  long cacheLastMod = -1;
  String cacheQueryString = null;
  String cachePathInfo = null;
  String cacheServletPath = null;
  Object lock = new Object();

  protected void service(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    // Utiliser le cache uniquement pour les requêtes GET.
    String method = req.getMethod();
    if (!method.equals("GET")) {
      super.service(req, res);
      return;
    }

    // Contrôler la date de dernière modification pour cette servlet.
    long servletLastMod = getLastModified(req);

    // -1 signifie que nous ne devons pas utiliser le cache.
    if (servletLastMod == -1) {
      super.service(req, res);
      return;
    }

    // Si le client a envoyé un en-tête If-Modified-Since égal ou supérieure
    // à la date de dernière modification de la servlet, envoyer un code
    // d'état "Not Modified". L'arrondi se fait à la seconde inférieure car
    // les en-têtes du client sont précis à la seconde près.
    if ((servletLastMod / 1000 * 1000) <=
            req.getDateHeader("If-Modified-Since")) {
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
      res.setStatus(res.SC_NOT_MODIFIED);
      return;
    }

    // Utiliser le cache existant s'il est valide.
    CacheHttpServletResponse localResponseCopy = null;
    synchronized (lock) {
      if (servletLastMod <= cacheLastMod &&
              cacheResponse.isValid() &&
              equal(cacheQueryString, req.getQueryString()) &&
              equal(cachePathInfo, req.getPathInfo()) &&
              equal(cacheServletPath, req.getServletPath())) {
        localResponseCopy = cacheResponse;
      }
    }
    if (localResponseCopy != null) {
      localResponseCopy.writeTo(res);
      return;
    }

    // Sinon, créer un nouveau cache pour capturer la réponse.
    localResponseCopy = new CacheHttpServletResponse(res);
    super.service(req, localResponseCopy);
    synchronized (lock) {
      cacheResponse = localResponseCopy;
      cacheLastMod = servletLastMod;
      cacheQueryString = req.getQueryString();
      cachePathInfo = req.getPathInfo();
      cacheServletPath = req.getServletPath();
    }
  }

  private boolean equal(String s1, String s2) {
    if (s1 == null && s2 == null) {
      return true;
    }
    else if (s1 == null || s2 == null) {
      return false;
    }
    else {
      return s1.equals(s2);
    }
  }
}

class CacheHttpServletResponse implements HttpServletResponse {
  // Stocker les variables clé de la réponse afin de les fixer plus tard.
  private int status;
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
  private Hashtable headers;
  private int contentLength;
  private String contentType;
  private Locale locale;
  private Vector cookies;
  private boolean didError;
  private boolean didRedirect;
  private boolean gotStream;
  private boolean gotWriter;

  private HttpServletResponse delegate;
  private CacheServletOutputStream out;
  private PrintWriter writer;

  CacheHttpServletResponse(HttpServletResponse res) {
    delegate = res;
    try {
      out = new CacheServletOutputStream(res.getOutputStream());
    }
    catch (IOException e) {
      System.out.println(
        "IOException reçue pendant la construction de la réponse dans "+
        "le cache : " + e.getMessage());
    }
    internalReset();
  }

  private void internalReset() {
    status = 200;
    headers = new Hashtable();
    contentLength = -1;
    contentType = null;
    locale = null;
    cookies = new Vector();
    didError = false;
    didRedirect = false;
    gotStream = false;
    gotWriter = false;
    out.getBuffer().reset();
  }

  public boolean isValid() {
    // Pas de cache pour les pages d'erreur ou les redirections.
    return didError != true && didRedirect != true;
  }

  private void internalSetHeader(String name, Object value) {
    Vector v = new Vector();
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
    v.addElement(value);
    headers.put(name, v);
  }

  private void internalAddHeader(String name, Object value) {
    Vector v = (Vector) headers.get(name);
    if (v == null) {
      v = new Vector();
    }
    v.addElement(value);
    headers.put(name, v);
  }

  public void writeTo(HttpServletResponse res) {
    // Écrire le code d'état.
    res.setStatus(status);
    // Écrire les en-têtes utilitaires.
    if (contentType != null) res.setContentType(contentType);
    if (locale != null) res.setLocale(locale);
    // Écrire les cookies.
    Enumeration enum = cookies.elements();
    while (enum.hasMoreElements()) {
      Cookie c = (Cookie) enum.nextElement();
      res.addCookie(c);
    }
    // Écrire les en-têtes standard.
    enum = headers.keys();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      Vector values = (Vector) headers.get(name);  // Peut avoir
      Enumeration enum2 = values.elements();       // plusieurs valeurs.
      while (enum2.hasMoreElements()) {
        Object value = enum2.nextElement();
        if (value instanceof String) {
          res.setHeader(name, (String)value);
        }
        if (value instanceof Integer) {
          res.setIntHeader(name, ((Integer)value).intValue());
        }
        if (value instanceof Long) {
          res.setDateHeader(name, ((Long)value).longValue());
        }
      }
    }
    // Écrire la longueur du contenu.
    res.setContentLength(out.getBuffer().size());
    // Écrire le corps.
    try {
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
      out.getBuffer().writeTo(res.getOutputStream());
    }
    catch (IOException e) {
      System.out.println(
        "IOException reçue pendant l'écriture de la réponse dans le cache: " +
        e.getMessage());
    }
  }

  public ServletOutputStream getOutputStream() throws IOException {
    if (gotWriter) {
      throw new IllegalStateException(
        "Impossible d'obtenir le flot de sortie après " +
        "avoir obtenu la sortie caractères");
    }
    gotStream = true;
    return out;
  }

  public PrintWriter getWriter() throws UnsupportedEncodingException {
    if (gotStream) {
      throw new IllegalStateException(
        "Impossible d'obtenir une sortie caractères après avoir obtenu "+
        "un flot de sortie");
    }
    gotWriter = true;
    if (writer == null) {
      OutputStreamWriter w =
        new OutputStreamWriter(out, getCharacterEncoding());
      writer = new PrintWriter(w, true);  // Vidage automatique nécessaire.
    }
    return writer;
  }

  public void setContentLength(int len) {
    delegate.setContentLength(len);
    // Pas besoin de sauvegarder la longueur ; nous pouvons la
    // calculer plus tard.
  }

  public void setContentType(String type) {
    delegate.setContentType(type);
    contentType = type;
  }

  public String getCharacterEncoding() {
    return delegate.getCharacterEncoding();
  }
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
  public void setBufferSize(int size) throws IllegalStateException {
    delegate.setBufferSize(size);
  }

  public int getBufferSize() {
    return delegate.getBufferSize();
  }

  public void reset() throws IllegalStateException {
    delegate.reset();
    internalReset();
  }

  public boolean isCommitted() {
    return delegate.isCommitted();
  }

  public void flushBuffer() throws IOException {
    delegate.flushBuffer();
  }

  public void setLocale(Locale loc) {
    delegate.setLocale(loc);
    locale = loc;
  }

  public Locale getLocale() {
    return delegate.getLocale();
  }

  public void addCookie(Cookie cookie) {
    delegate.addCookie(cookie);
    cookies.addElement(cookie);
  }

  public boolean containsHeader(String name) {
    return delegate.containsHeader(name);
  }

  /** @deprecated */
  public void setStatus(int sc, String sm) {
    delegate.setStatus(sc, sm);
    status = sc;
  }

  public void setStatus(int sc) {
    delegate.setStatus(sc);
    status = sc;
  }
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
public void setHeader(String name, String value) {
  delegate.setHeader(name, value);
  internalSetHeader(name, value);
}

public void setIntHeader(String name, int value) {
  delegate.setIntHeader(name, value);
  internalSetHeader(name, new Integer(value));
}

public void setDateHeader(String name, long date) {
  delegate.setDateHeader(name, date);
  internalSetHeader(name, new Long(date));
}

public void sendError(int sc, String msg) throws IOException {
  delegate.sendError(sc, msg);
  didError = true;
}

public void sendError(int sc) throws IOException {
  delegate.sendError(sc);
  didError = true;
}

public void sendRedirect(String location) throws IOException {
  delegate.sendRedirect(location);
  didRedirect = true;
}

public String encodeURL(String url) {
  return delegate.encodeURL(url);
}

public String encodeRedirectURL(String url) {
  return delegate.encodeRedirectURL(url);
}

public void addHeader(String name, String value) {
  internalAddHeader(name, value);
}

public void addIntHeader(String name, int value) {
  internalAddHeader(name, new Integer(value));
}

public void addDateHeader(String name, long value) {
  internalAddHeader(name, new Long(value));
}
```

Exemple 3-11. La classe CacheHttpServlet (suite)

```
  /** @deprecated */
  public String encodeUrl(String url) {
    return this.encodeURL(url);
  }

  /** @deprecated */
  public String encodeRedirectUrl(String url) {
    return this.encodeRedirectURL(url);
  }
}

class CacheServletOutputStream extends ServletOutputStream {

  ServletOutputStream delegate;
  ByteArrayOutputStream cache;

  CacheServletOutputStream(ServletOutputStream out) {
    delegate = out;
    cache = new ByteArrayOutputStream(4096);
  }

  public ByteArrayOutputStream getBuffer() {
    return cache;
  }

  public void write(int b) throws IOException {
    delegate.write(b);
    cache.write(b);
  }

  public void write(byte b[]) throws IOException {
    delegate.write(b);
    cache.write(b);
  }

  public void write(byte buf[], int offset, int len) throws IOException {
    delegate.write(buf, offset, len);
    cache.write(buf, offset, len);
  }
}
```

4

Retrouver l'information

Pour construire une application web réussie, il est souvent nécessaire d'avoir beaucoup d'informations sur l'environnement dans lequel elle s'exécute. Vous pourrez avoir besoin d'informations à propos du serveur qui exécute vos servlets et sur les spécificités du client qui soumet les requêtes. Et peu importe l'environnement dans lequel s'exécute l'application, vous aurez certainement besoin d'informations sur la requête traitée.

Les servlets possèdent un certain nombre de méthodes permettant d'avoir accès à ces informations. En général, chaque méthode retourne un résultat spécifique. Si vous comparez cela à la façon dont les variables d'environnement sont utilisées pour passer des informations à un programme CGI, l'approche servlet a de nombreux avantages :

- Contrôle de type fort. Autrement dit, une aide plus importante apportée par le compilateur pour la détection des erreurs. Un programme CGI utilise une fonction pour retrouver ses variables d'environnement. Beaucoup d'erreurs ne seront trouvées qu'au moment où elles provoquent un problème à l'exécution. Regardons comment un programme CGI et une servlet déterminent le port du serveur sur lequel ils s'exécutent.

  Un script CGI écrit en Perl appelle :

  ```
  $port = $ENV{'SERVER_PORT'};
  ```

  où `$port` est une variable non typée. Un programme CGI écrit en C appelle :

  ```
  char *port = getenv("SERVER_PORT");
  ```

  où `port` est un pointeur sur une chaîne de caractères. Il y a de grandes chances qu'il y ait des erreurs accidentelles. Le nom de la variable d'environnement pourrait être mal orthographié (cela arrive souvent) ou le type de données pourrait ne pas correspondre à celui retourné par la variable d'environnement.

  D'un autre côté, une servlet appelle :

  ```
  int port = req.getServerPort()
  ```

  Cela élimine beaucoup d'erreurs accidentelles car le compilateur peut garantir qu'il n'y a pas d'erreur d'orthographe et que chaque type de retour est ce qu'il doit être.

- Report de calcul. Quand un serveur lance un programme CGI, la valeur de chaque variable d'environnement doit être pré-calculée et passée, que le programme les

utilise ou non. Un serveur lançant une servlet peut améliorer les performances en effectuant ces calculs à la demande.

- Plus d'interaction avec le serveur. Une fois l'exécution du programme CGI commencée, celui-ci est détaché de son serveur. Le seul chemin de communication disponible est la sortie standard. En revanche, une servlet peut coopérer avec le serveur. Comme nous l'avons vu dans le chapitre précédent, une servlet coopère soit en faisant partie du serveur (lorsque c'est possible), soit en étant dans un processus extérieur connecté au serveur (lorsque c'est nécessaire). En utilisant cette connectivité, une servlet peut effectuer des requêtes *ad hoc* afin d'obtenir des informations que seul le serveur peut fournir. Par exemple, une servlet peut demander au serveur de réaliser des traductions de chemin arbitraires, prenant en compte les alias du serveur et les chemins virtuels.

Si vous venez du monde CGI, le *tableau 4-1* peut vous aider à effectuer votre migration. Il donne la liste de chaque variable d'environnement CGI et la méthode de servlet HTTP correspondante.

Tableau 4-1. Les variables d'environnement CGI et les méthodes de servlet correspondantes

Variable d'environnement CGI	Méthodes d'une servlet HTTP
SERVER_NAME	`req.getServerName()`
SERVER_SOFTWARE	`getServletContext().getServerInfo()`
SERVER_PROTOCOL	`req.getProtocol()`
SERVER_PORT	`req.getServerPort()`
REQUEST_METHOD	`req.getMethod()`
PATH_INFO	`req.getPathInfo()`
PATH_TRANSLATED	`req.getPathTranslated()`
SCRIPT_NAME	`req.getServletPath()`
DOCUMENT_ROOT	`getServletContext().getRealPath("/")`
QUERY_STRING	`req.getQueryString()`
REMOTE_HOST	`req.getRemoteHost()`
REMOTE_ADDR	`req.getRemoteAddr()`
AUTH_TYPE	`req.getAuthType()`
REMOTE_USER	`req.getRemoteUser()`
CONTENT_TYPE	`req.getContentType()`
CONTENT_LENGTH	`req.getContentLength()`
HTTP_ACCEPT	`req.getHeader("Accept")`
HTTP_USER_AGENT	`req.getHeader("User-Agent")`
HTTP_REFERER	`req.getHeader("Referer")`

Dans le reste de ce chapitre, nous verrons comment et quand utiliser ces méthodes, ainsi que plusieurs autres méthodes qui n'ont pas d'équivalent CGI. Au fur et à mesure, nous mettrons ces méthodes en pratique dans des servlets réelles.

La servlet

Chaque nom de servlet enregistré dispose de paramètres d'initialisation spécifiques. Ces paramètres sont disponibles à la servlet à tout moment ; ils sont définis dans le descripteur de déploiement *web.xml* et souvent utilisés dans `init()` pour définir des valeurs initiales ou par défaut de la servlet, ou pour personnaliser son comportement d'une certaine façon. Les paramètres d'initialisation sont expliqués au chapitre 3, *Cycle de vie d'une servlet*.

Obtenir un paramètre d'initialisation de la servlet

Une servlet utilise la méthode `getInitParameter()` pour avoir accès à ses paramètres d'initialisation :

```
public String ServletConfig.getInitParameter(String name)
```

Cette méthode retourne la valeur du paramètre indiqué ou `null` s'il n'existe pas. La valeur de retour est toujours un `String`. C'est à la servlet d'interpréter la valeur.

La classe `GenericServlet` implémente l'interface `ServletConfig`, fournissant ainsi un accès direct à la méthode `getInitParameter()`. Elle est généralement utilisée de la façon suivante :

```
public void init() throws ServletException {
  String greeting = getInitParameter("greeting");
}
```

Une servlet devant établir une connexion à une base de données peut utiliser les paramètres d'initialisation pour en définir les détails. Nous pouvons imaginer une méthode `establishConnection()` personnalisée pour faire abstraction des détails de JDBC, comme le montre l'*exemple 4-1*.

Exemple 4-1. Utiliser les paramètres d'initialisation pour établir une connexion à une base de données

```
java.sql.Connection con = null;

public void init() throws ServletException {
  String host = getInitParameter("host");
  int port = Integer.parseInt(getInitParameter("port"));
  String db = getInitParameter("db");
  String user = getInitParameter("user");
  String password = getInitParameter("password");
  String proxy = getInitParameter("proxy");

  con = establishConnection(host, port, db, user, password, proxy);
}
```

Les servlets conçues pour J2EE (*Java 2, Enterprise Edition*) ont à leur disposition d'autres modèles d'abstractions standard et plus avancés. Voir le chapitre 12, *Servlets d'entreprise et J2EE*.

Obtenir le nom des paramètres d'initialisation de la servlet

Une servlet peut examiner tous ses paramètres d'initialisation en utilisant `getInitParameterNames()` :

```
public Enumeration ServletConfig.getInitParameterNames()
```

Cette méthode retourne le nom de tous les paramètres d'initialisation de la servlet sous forme d'un `Enumeration` d'objets `String` ou d'un `Enumeration` vide s'il n'existe aucun paramètre. Elle est le plus souvent utilisée pour le débogage.

La classe `GenericServlet` fournit cette méthode aux servlets. L'*exemple 4-2* montre une servlet qui imprime le nom et la valeur de tous ses paramètres d'initialisation

Exemple 4-2. Obtenir le nom des paramètres d'initialisation

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class InitSnoop extends GenericServlet {

  // La méthode init() n'est pas nécessaire.

  public void service(ServletRequest req, ServletResponse res)
                           throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    out.println("Paramètres d'initialisation :");
    Enumeration enum = getInitParameterNames();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      out.println(name + " : " + getInitParameter(name));
    }
  }
}
```

Notez que cette servlet étend directement la classe `GenericServlet`, montrant que les paramètres d'initialisation sont disponibles aux servlets qui ne sont pas des servlets HTTP. Une servlet générique peut être utilisée dans un serveur web même s'il lui manque le support des fonctionnalités spécifiques à HTTP.

Obtenir le nom d'une servlet

Il existe aussi dans l'interface `ServletConfig` une méthode qui retourne le nom enregistré de la servlet :

```
public String ServletConfig.getServletName()
```

Si la servlet n'est pas enregistrée, cette méthode revoie le nom de classe de la servlet. Cette méthode se révèle utile pour l'écriture de journaux (*logs*) et pour le stockage des in-

formations d'état d'une instance de servlet dans une ressource partagée comme une base de données ou le `SessionContext` de la servlet que nous verrons un peu plus loin.

Le code suivant montre comment utiliser le nom de la servlet pour retrouver une valeur dans le contexte de servlet, le nom faisant partie de la clé de recherche :

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
  String name = getServletName();
  ServletContext context = getServletContext();
  Object value = context.getAttribute(name + ".state");
}
```

En utilisant le nom de la servlet dans la clé, chaque instance de servlet peut facilement garder une valeur d'attribut séparée dans un contexte partagé.

Le serveur

Une servlet peut en apprendre beaucoup sur son serveur. En autres choses, elle peut en connaître le nom de machine, le port d'écoute et le logiciel serveur. Ces informations peuvent être fournies au client, elle peut les utiliser pour personnaliser son comportement en fonction d'un paquetage particulier du serveur ou même les utiliser explicitement pour limiter les machines sur lesquelles elle s'exécute.

Obtenir des informations sur le serveur

Une servlet obtient la plupart des informations sur le serveur au travers de son objet `ServletContext`. Avant l'API 2.2, le `ServletContext` était généralement vu comme une référence au serveur lui-même. Depuis l'API 2.2, les règles ont changé et il y a maintenant un `ServletContext` différent pour chaque application web s'exécutant sur le serveur. `ServletContext` est devenu une référence à l'application web et non au serveur. Pour des requêtes simples, il y a peu de différences.

Une servlet dispose de cinq méthodes pour en apprendre plus sur son serveur : deux sont appelées en utilisant l'objet `ServletRequest` passé à la servlet et trois autres sont appelées à partir de l'objet `ServletContext` dans lequel la servlet s'exécute.

Une servlet peut obtenir le nom du serveur et le numéro de port pour une requête particulière avec `getServerName()` et `getServerPort()`, respectivement :

```
public String ServletRequest.getServerName()
public int ServletRequest.getServerPort()
```

Ces méthodes sont des attributs de `ServletRequest` car les valeurs peuvent changer avec les requêtes si le serveur a plus d'un nom (une technique appelée *hébergement virtuel*). Le nom retourné pourrait ressembler à `www.servlets.com` et le port à `8080`.

Les méthodes `getServerInfo()` et `getAttribute()` de `ServletContext` fournissent des informations sur le logiciel serveur et ses attributs :

```
public String ServletContext.getServerInfo()
public Object ServletContext.getAttribute(String name)
```

`getServerInfo()` retourne le nom et la version du logiciel serveur, séparés par une barre oblique (`/`). La chaîne retournée pourrait ressembler à `Tomcat Web Server/3.2`. Cer-

tains serveurs ajoute à la fin des informations supplémentaire décrivant l'environnement de fonctionnement du serveur.

`getAttribute()` retourne la valeur de l'attribut de serveur précisé sous forme d'un `Object` ou `null` si l'attribut n'existe pas. Les serveurs ont la possibilité de figer des attributs dans le contexte utilisé par les servlets. Vous pouvez voir cette méthode comme un chemin détourné par lequel une servlet peut obtenir des informations supplémentaires sur son serveur. Par exemple, un serveur peut proposer des statistiques sur sa charge, une référence à une réserve de ressources partagée ou toute autre information potentiellement utile. Le seul attribut obligatoire est l'attribut `javax.servlet.context.tempdir` qui fournit une référence `java.io.File` à un répertoire privé à ce contexte.

Les servlets peuvent aussi ajouter leurs propres attributs au contexte à l'aide de la méthode `setAttribute()`, comme expliqué au chapitre 11, *Collaboration de servlets*. Les noms d'attributs doivent suivre la même convention de nommage que celle des paquetages. Les noms de paquetage `java.*` et `javax.*` sont réservés à la division Java Software de Sun Microsystems (connue sous le nom JavaSoft), et `com.sun.*` est réservé à Sun Microsystems. Reportez-vous à la documentation de votre serveur pour avoir la liste de ses attributs. La liste de tous les attributs stockés par le serveur et les autres servlets s'obtient à l'aide de `getAttributeNames()` :

```
public Enumeration ServletContext.getAttributeNames()
```

Ces méthodes étant des attributs du `ServletContext` dans lequel la servlet s'exécute, vous devez les appeler au travers de cet objet :

```
String serverInfo = getServletContext().getServerInfo();
```

L'utilisation la plus évidente de ces informations sur le serveur est pour une servlet « À propos du serveur », comme montré à l'*exemple 4-3*.

Exemple 4-3. Examiner le serveur

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class ServerSnoop extends GenericServlet {

  public void service(ServletRequest req, ServletResponse res)
                          throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    ServletContext context = getServletContext();
    out.println("req.getServerName() : " + req.getServerName());
    out.println("req.getServerPort() : " + req.getServerPort());
    out.println("context.getServerInfo() : " + context.getServerInfo());
    out.println("nom de getServerInfo() : " +
                getServerInfoName(context.getServerInfo()));
    out.println("version de getServerInfo() : " +
                getServerInfoVersion(context.getServerInfo()));
    out.println("context.getAttributeNames() :");
    Enumeration enum = context.getAttributeNames();
```

Exemple 4-3. Examiner le serveur (suite)

```
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      out.println("  context.getAttribute(\"" + name + "\") : " +
                  context.getAttribute(name));
    }
  }

  private String getServerInfoName(String serverInfo) {
    int slash = serverInfo.indexOf('/');
    if (slash == -1) return serverInfo;
    else return serverInfo.substring(0, slash);
  }

  private String getServerInfoVersion(String serverInfo) {
    // L'information de version se trouve entre le / et l'espace.
    int slash = serverInfo.indexOf('/');
    if (slash == -1) return null;
    int space = serverInfo.indexOf(' ', slash);
    if (space == -1) space = serverInfo.length();
    return serverInfo.substring(slash + 1, space);
  }
}
```

Cette servlet étend aussi directement GenericServlet, montrant que toutes les informations concernant un serveur sont disponibles aux servlets de tout type. La sortie de la servlet est du texte simple, dont voici un exemple :

```
req.getServerName() : localhost
req.getServerPort() : 8080
context.getServerInfo() : Tomcat Web Server/3.2 (JSP 1.1; Servlet 2.2; ...)
nom de getServerInfo() : Tomcat Web Server
version de getServerInfo() : 3.2
context.getAttributeNames() :
  context.getAttribute("javax.servlet.context.tempdir"):
work/localhost_8080
```

Écrire dans un fichier temporaire

L'attribut `javax.servlet.context.tempdir` désigne un répertoire temporaire dans lequel des fichiers à durée de vie limitée peuvent être stockés. Chaque contexte reçoit un répertoire temporaire différent. Pour l'exemple précédent, le répertoire est *racine_serveur/work/localhost_8080*. L'*exemple 4-4* montre comment écrire un fichier temporaire dans ce répertoire temporaire.

Exemple 4-4. Créer un fichier temporaire dans le répertoire temporaire

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                  throws ServletException, IOException {
  // Le répertoire est donné sous forme d'un objet File.
```

Exemple 4-4. Créer un fichier temporaire dans le répertoire temporaire (suite)

```
  File dir = (File) getServletContext()
                    .getAttribute("javax.servlet.context.tempdir");

  // Construire un fichier temporaire dans le répertoire temporaire
  // (méthode du JDK 1.2).
  File f = File.createTempFile("xxx", ".tmp", dir);

  // Se préparer à écrire dans le fichier.
  FileOutputStream fout = new FileOutputStream(f);

  // ...
}
```

La servlet commence par localiser le répertoire temporaire. Ensuite, elle utilise la méthode `createTempFile()` pour y créer un fichier temporaire avec un préfixe xxx et une extension `.tmp`. Enfin, elle construit un `FileOutputStream` afin d'écrire dans le fichier temporaire. Les fichiers qui doivent persister entre les redémarrages du serveur ne doivent pas être placés dans le répertoire temporaire.

Verrouiller une servlet à un serveur

Les informations sur le serveur peuvent être utilisées de façon plus productive. Supposons que vous écriviez une servlet et que vous ne souhaitiez pas qu'elle puisse être exécutée n'importe où. Peut-être voulez-vous la vendre et, pour limiter les copies illégales, voulez-vous la lier à la machine de votre client par une licence logicielle. Ou vous avez écrit un générateur de licences sous forme de servlet et vous voulez vous assurer qu'il fonctionne uniquement derrière votre firewall. Cela peut se faire assez facilement car une servlet a un accès instantané aux informations sur son serveur.

L'*exemple 4-5* montre une servlet qui se lie elle-même à une adresse IP et à un numéro de port de serveur particuliers. Elle demande un paramètre d'initialisation, key, approprié à l'adresse IP et au port du serveur avant de se déverrouiller et de traiter une requête. Si elle ne reçoit pas la clé appropriée, elle refuse de continuer. L'algorithme utilisé pour faire correspondre la clé à une adresse IP et un port (et vice versa) doit être sécurisé.

Exemple 4-5. Une servlet verrouillée à un serveur

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;

public class KeyedServerLock extends GenericServlet {

  // Cette servlet n'a pas de variables de classe ou d'instance associées
  // à ce verrouillage, de manière à simplifier les problèmes de
  // synchronisation.
```

Exemple 4-5. Une servlet verrouillée à un serveur (suite)

```
public void service(ServletRequest req, ServletResponse res)
                          throws ServletException, IOException {
  res.setContentType("text/plain");
  PrintWriter out = res.getWriter();

  // Le test de piratage ne devrait pas se faire dans init() car
  // name/port font partie de la requête.
  String key = getInitParameter("key");
  String host = req.getServerName();
  int port = req.getServerPort();

  // Vérifier si le paramètre "key" débloque ce serveur.
  if (! keyFitsServer(key, host, port)) {
    // Expliquer, condamner, menacer, etc.
    out.println("Pirate !");
  }
  else {
    // Servir.
    out.println("Valide");
    // etc...
  }
}

// Cette méthode contient l'algorithme utilisé pour faire correspondre
// une clé à un nom et un numéro de machine. Cette implémentation est
// très faible et ne doit pas être utilisée par des sites commerciaux.
//
private boolean keyFitsServer(String key, String host, int port) {

  if (key == null) return false;

  long numericKey = 0;
  try {
    numericKey = Long.parseLong(key);
  }
  catch (NumberFormatException e) {
    return false;
  }

  // La clé doit être un nombre sur 64 bits égal au non logique (~) d'une
  // adresse IP sur 32 bits concaténée au numéro de port sur 32 bits.

  byte hostIP[];
  try {
    hostIP = InetAddress.getByName(host).getAddress();
  }
  catch (UnknownHostException e) {
    return false;
  }
```

Exemple 4-5. Une servlet verrouillée à un serveur (suite)

```
    // Obtenir l'adresse IP sur 32 bits.
    long servercode = 0;
    for (int i = 0; i < 4; i++) {
      servercode <<= 8;
      servercode |= hostIP[i];
    }

    // Concaténer le numéro de port sur 32 bits.
    servercode <<= 32;
    servercode |= port;

    // Non logique.
    long accesscode = ~numericKey;

    // Le moment de vérité : la clé est-elle bonne ?
    return (servercode == accesscode);
  }
}
```

Cette servlet refuse de fonctionner sans clé correcte. Cependant, afin de la rendre sûre, la logique simple de `keyFitsServer()` doit être remplacée par un algorithme plus fiable et la servlet entière doit être passée dans un brouilleur afin d'empêcher sa décompilation. L'*exemple 4-13* donne le code utilisé pour générer des clés. Si vous utilisez cette servlet vous-même, il est préférable d'accéder au serveur sous son vrai nom, au lieu de `localhost`, de façon à ce que la servlet puisse déterminer le vrai nom du serveur web et son adresse IP.

Obtenir un paramètre d'initialisation du contexte

Les paramètres d'initialisation de la servlet, comme présentés précédemment, sont passés à chaque servlet. Quand plusieurs servlets doivent recevoir les mêmes valeurs d'initialisation, ces valeurs peuvent être données comme un paramètre d'initialisation du *contexte*. La classe `ServletContext` a deux méthodes, `getInitParameter()` et `getInitParameterNames()`, permettant de retrouver des informations d'initialisation dans le contexte :

```
public String ServletContext.getInitParameter(String name)
public Enumeration ServletContext.getInitParameterNames()
```

Ces méthodes sont similaires à leurs contreparties dans `ServletConfig`. `getInitParameter(String name)` renvoie la valeur chaîne du paramètre indiqué. `getInitParameterNames()` retourne un `Enumeration` contenant le nom de tous les paramètres disponibles à l'application web, ou un `Enumeration` vide s'il n'y a aucun paramètre.

Les paramètres d'initialisation d'un contexte sont spécifiés dans le descripteur de déploiement *web.xml* du contexte à l'aide de la balise `<context-param>` (voir l'*exemple 4-6*).

Exemple 4-6. Fixer des paramètres d'initialisation du contexte dans le descripteur de déploiement

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <context-param>
        <param-name>
            rmihost
        </param-name>
        <param-value>
            localhost
        </param-value>
    </context-param>
    <context-param>
        <param-name>
            rmiport
        </param-name>
        <param-value>
            1099
        </param-value>
    </context-param>
</web-app>
```

Toute servlet présente dans cette application web peut lire les paramètres d'initialisation du contexte pour localiser le registre RMI partagé, comme le montre l'*exemple 4-7*.

Exemple 4-7. Trouver le registre à l'aide des paramètres d'initialisation du contexte

```
import java.io.*;
import java.rmi.registry.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RmiDemo extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    try {
      ServletContext context = getServletContext();
      String rmihost = context.getInitParameter("rmihost");
      int rmiport = Integer.parseInt(context.getInitParameter("rmiport"));

      Registry registry = LocateRegistry.getRegistry(rmihost, rmiport);
```

Exemple 4-7. Trouver le registre à l'aide des paramètres d'initialisation du contexte (suite)

```
      // ...
    }
    catch (Exception e) {
      // ...
    }
  }
}
```

Il n'existe pas de mécanisme standard pour créer des paramètres d'initialisation globaux visibles de tous les contextes.

Déterminer la version de l'API Servlet

Une servlet peut aussi demander au serveur la version de l'API Servlet qu'il supporte. En dehors de son utilité pour le débogage, une servlet peut exploiter cette information pour décider d'employer une nouvelle manière, ou une plus ancienne peut-être moins efficace, de résoudre un problème. L'API Servlet 2.1 a introduit deux méthodes qui retournent des informations de version :

```
public int ServletContext.getMajorVersion()
public int ServletContext.getMinorVersion()
```

Pour l'API 2.1, `getMajorVersion()` retourne `2` et `getMinorVersion()` retourne `1`. Bien entendu, ces méthodes fonctionnent uniquement pour les servlets qui s'exécutent dans des serveurs supportant l'API Servlet 2.1 et ultérieure. Pour déterminer la version courante de l'API Servlet sur tous les serveurs, vous pouvez vous servir de `com.oreilly.servlet.VersionDetector`. Cette classe ne demande pas la version au serveur ; elle recherche les classes et les variables disponibles dans l'environnement d'exécution et elle se base sur sa connaissance de l'historique de l'API Servlet pour calculer la version courante, de 1.0 à 2.3. Étant donné que la classe n'appelle ni `getMajorVersion()` ni `getMinorVersion()`, elle fonctionne et se compile sur toutes les versions de l'API. La classe `VersionDetector` détermine aussi la version courante du JDK, de 1.0 à 1.3, en utilisant la même technique. Il s'avère que cette méthode est plus fiable sur toutes les implémentations de la JVM que d'interroger la classe `System`. L'*exemple 4-8* montre la classe `VersionDetector`. Les mises à jour de cette classe pour supporter les futures versions de l'API Servlet et du JDK seront postées sur *http://www.servlets.com*.

Exemple 4-8. La classe VersionDetector

```
package com.oreilly.servlet;

public class VersionDetector {

  static String servletVersion;
  static String javaVersion;

  public static String getServletVersion() {
    if (servletVersion != null) {
      return servletVersion;
    }
```

Exemple 4-8. La classe VersionDetector (suite)

```
    // javax.servlet.http.HttpSession est arrivé avec l'API Servlet 2.0.
    // javax.servlet.RequestDispatcher est arrivé avec l'API Servlet 2.1.
    // javax.servlet.http.HttpServletResponse.SC_EXPECTATION_FAILED est
    //     arrivé avec l'API Servlet 2.2.
    // javax.servlet.Filter est prévu dans l'API Servlet 2.3.
    String ver = null;
    try {
      ver = "1.0";
      Class.forName("javax.servlet.http.HttpSession");
      ver = "2.0";
      Class.forName("javax.servlet.RequestDispatcher");
      ver = "2.1";
      Class.forName("javax.servlet.http.HttpServletResponse")
                   .getDeclaredField("SC_EXPECTATION_FAILED");
      ver = "2.2";
      Class.forName("javax.servlet.Filter");
      ver = "2.3";
    } catch (Throwable t) {
    }

    servletVersion = ver;
    return servletVersion;
  }

  public static String getJavaVersion() {
    if (javaVersion != null) {
      return javaVersion;
    }

    // java.lang.Void est arrivé avec le JDK 1.1.
    // java.lang.ThreadLocal est arrivé avec le JDK 1.2.
    // java.lang.StrictMath est arrivé avec le JDK 1.3.
    String ver = null;
    try {
      ver = "1.0";
      Class.forName("java.lang.Void");
      ver = "1.1";
      Class.forName("java.lang.ThreadLocal");
      ver = "1.2";
      Class.forName("java.lang.StrictMath");
      ver = "1.3";
    } catch (Throwable t) {
    }

    javaVersion = ver;
    return javaVersion;
  }
}
```

La classe essaie de charger des classes et d'accéder à des variables jusqu'à ce qu'une exception `NoClassDefFoundError` ou `NoSuchFieldException` stoppe la recherche. À ce moment-là, la version courante est connue. L'*exemple 4-9* montre une servlet qui examine la version de l'API Servlet et du JDK.

Exemple 4-9. Examiner les versions

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.VersionDetector;

public class VersionSnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                      throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    out.println("Version de l'API Servlet : " +
        VersionDetector.getServletVersion());
    out.println("Version du JDK : " +
        VersionDetector.getJavaVersion());
  }
}
```

Le client

Pour chaque requête, une servlet a la possibilité d'obtenir des informations sur la machine cliente et, pour les pages demandant une authentification, sur l'utilisateur. Ces informations peuvent être utilisées pour la journalisation des données de connexion, pour associer des informations avec des utilisateurs individuels ou pour restreindre les accès à certains clients.

Obtenir des informations sur la machine du client

Une servlet utilise `getRemoteAddr()` et `getRemoteHost()` pour retrouver l'adresse IP et le nom de la machine du client, respectivement :

```
public String ServletRequest.getRemoteAddr()
public String ServletRequest.getRemoteHost()
```

Les deux valeurs sont retournées sous forme de `String`. L'information provient de la socket qui connecte le client et le serveur, l'adresse distante et le nom de machine peuvent être ceux d'un serveur proxy. Un exemple d'adresse distante pourrait être `192.26.80.118` et celui d'une machine distante pourrait être `dist.engr.sgi.com`.

L'adresse IP et le nom distant peuvent être convertis en un objet `java.net.InetAddress` par `InetAddress.getByName()` :

```
InetAddress remoteInetAddress = InetAddress.getByName(req.getRemoteAddr());
```

Restreindre l'accès

Du fait de la politique du gouvernement des États-Unis sur l'exportation des outils de chiffrement fort, certains sites web doivent faire attention à qui télécharge certains logiciels. Les servlets, par leur possibilité d'obtenir des informations sur la machine du client, sont très bien adaptées à garantir cette restriction. Elles peuvent contrôler la machine du client et ne fournir les liens pour le téléchargement que s'il vient d'un pays autorisé.

Dans la première édition de ce livre, les seuls pays autorisés étaient les États-Unis et le Canada, et cette servlet était écrite de manière à n'autoriser les téléchargements qu'aux utilisateurs venant de ce deux pays. Depuis, le gouvernement des États-Unis a assoupli sa politique sur l'exportation des outils de chiffrement et, aujourd'hui, la plupart des logiciels de chiffrement peuvent être téléchargés par tout le monde, excepté ceux habitants dans les pays faisant partie de la liste « Terrorist 7 » : Cuba, Iran, Iraq, Corée du nord, Lybie, Syrie et Soudan. L'*exemple 4-10* montre une servlet qui autorise les téléchargements à toute personnes n'habitant pas dans ces sept pays.

Exemple 4-10. Peut-on leur faire confiance ?

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ExportRestriction extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // ...Du HTML d'introduction...

    // Récupérer le nom de la machine du client.
    String remoteHost = req.getRemoteHost();

    // Voir si le client est autorisé.
    if (! isHostAllowed(remoteHost)) {
      out.println("Accès <BLINK>refusé</BLINK>");
    }
    else {
      out.println("Accès autorisé");
      // Afficher les liens de téléchargement, etc...
    }
  }

  // Interdire les noms de machine se terminant par .cu, .ir, .iq, .kp, .ly,
  // .sy et .sd.
```

Exemple 4-10. Peut-on leur faire confiance ? (suite)

```
  private boolean isHostAllowed(String host) {
    return (!host.endsWith(".cu") &&
            !host.endsWith(".ir") &&
            !host.endsWith(".iq") &&
            !host.endsWith(".kp") &&
            !host.endsWith(".ly") &&
            !host.endsWith(".sy") &&
            !host.endsWith(".sd"));
  }
}
```

Cette servlet obtient le nom de la machine du client par un appel à `req.getRemoteHost()` et, en fonction du suffixe, décide si le client vient d'un pays interdit ou non. Bien entendu, assurez-vous d'avoir eu de bons conseils légaux avant de rendre disponible en téléchargement tout code de cryptographie.

Obtenir des informations sur l'utilisateur

Que faites-vous lorsque vous devez restreindre l'accès à certaines de vos page web mais que vous souhaitez un contrôle plus fin que la restriction continent par continent ? Par exemple, disons que vous publiez un magazine en ligne et que seuls les abonnés payants peuvent lire les articles. Bien (préparez-vous), il n'y a pas besoin des servlets pour cela.

Presque tous les serveurs HTTP intègrent des possibilités permettant de restreindre l'accès à certaines ou à toutes les pages à un ensemble donné d'utilisateurs enregistrés. La façon de mettre en place l'accès restreint est décrite au chapitre 8, *Sécurité*, mais voici comment la mécanique fonctionne. La première fois qu'un navigateur tente d'accéder à une de ces pages, le serveur HTTP répond qu'il a besoin d'une authentification de l'utilisateur. Lorsque le navigateur reçoit cette réponse, il ouvre généralement une fenêtre demandant à l'utilisateur d'entrer un nom et un mot de passe afin d'accéder à la page, comme le montre la *figure 4-1*.

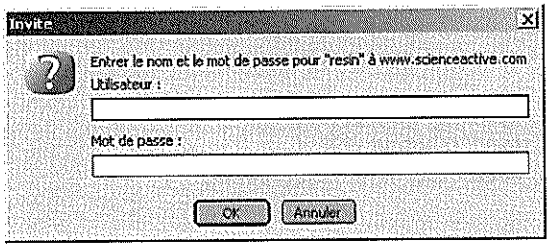


Figure 4-1. Fenêtre d'ouverture de connexion à une page à accès restreint

Une fois cette information entrée, le navigateur essaie à nouveau d'accéder à la page, en attachant à la requête le nom et le mot de passe de l'utilisateur. Si le serveur accepte la paire nom/mot de passe, il traite la requête. Si, en revanche, le serveur ne l'accepte pas, il refuse l'accès au navigateur et l'utilisateur regrette d'avoir encore oublié un autre mot de passe.

En quoi cela concerne-t-il les servlets ? Lorsque l'accès à une servlet a été restreint par le serveur, la servlet peut obtenir le nom de l'utilisateur accepté par le serveur, en utilisant la méthode `getRemoteUser()` :

```
public String HttpServletRequest.getRemoteUser()
```

Notez que cette information est retrouvée à partir de l'objet `HttpServletRequest` de la servlet, la sous-classe spécifique à HTTP de `ServletRequest`. Cette méthode retourne le nom de l'utilisateur effectuant la requête sous forme d'un `String` ou `null` si l'accès à la servlet n'a pas été restreint. Il n'y a pas de méthode équivalente permettant d'obtenir le mot de passe de l'utilisateur distant (bien qu'il puisse être déterminé manuellement, comme le montre l'*exemple 8-2* au chapitre 8). Un exemple d'utilisateur distant pourrait être `jhunter`.

Une servlet peut aussi utiliser la méthode `getAuthType()` pour déterminer le type d'autorisation qui a été utilisé.

```
public String HttpServletRequest.getAuthType()
```

Cette méthode retourne le type d'autorisation utilisé ou `null` si l'accès à la servlet n'a pas été restreint. Les types classiques d'autorisation sont `BASIC`, `DIGEST`, `FORM` et `CLIENT-CERT`. Consultez le chapitre 8 pour plus d'informations sur les différents types d'authentification.

Au moment où la servlet appelle `getRemoteUser()`, le serveur a déjà déterminé que l'utilisateur est autorisé à invoquer la servlet, mais cela ne veut pas dire que le nom de l'utilisateur distant est inutile. La servlet peut effectuer une seconde vérification d'autorisation, plus restrictive et dynamique que celle du serveur. Par exemple, elle peut retourner des informations sensibles sur quelqu'un seulement si c'est cette même personne qui effectue la requête, ou elle peut garantir une règle comme « chaque utilisateur ne peut effectuer que 10 accès par jour »[1].

Le nom du client ne fait qu'indiquer qui accède à la servlet. Après tout, la machine distante n'est pas nécessairement réservée à un seul client. Les serveurs Unix abritent le plus souvent des centaines d'utilisateurs et des proxies peuvent servir de passerelle à des milliers. Gardez bien à l'esprit que l'accès au nom du client a un prix. Chaque utilisateur doit être enregistré dans votre serveur et, avant d'accéder à votre site, il doit donner son nom et son mot de passe. De façon générale, l'authentification ne devrait être utilisée que si une servlet peut déterminer à qui elle parle. Le chapitre 7, *Suivi de session*, décrit de meilleures techniques et nécessitant une maintenance moindre, qui permettent de connaître les utilisateurs. Cependant, si une servlet est déjà protégée et que le nom lui est facilement accessible, elle peut l'utiliser.

Avec le nom de l'utilisateur distant, une servlet peut sauvegarder des informations sur chaque client. Sur le long terme, elle peut se rappeler les préférences de chaque individu. Sur le court terme, elle peut se rappeler des pages vues par le client et les utiliser pour ajouter une forme d'état sur un protocole HTTP sans état. Le suivi de session décrit au chapitre 7 peut être inutile si la servlet connaît déjà le nom du client.

---

1. Voulez-vous savoir comment dire « accès refusé » au onzième accès ? C'est dans le prochain chapitre.

Un accueil personnalisé

Une servlet simple utilisant `getRemoteUser()` peut accueillir ses clients par leur nom et se souvenir de leur dernière connexion, comme le montre l'*exemple 4-11*.

Exemple 4-11. Hé, mais je me souviens de vous !

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PersonalizedWelcome extends HttpServlet {

  Hashtable accesses = new Hashtable();

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // ...Du HTML d'introduction...

    String remoteUser = req.getRemoteUser();

    if (remoteUser == null) {
      out.println("Welcome!");
    }
    else {
      out.println("Bonjour, " + remoteUser + "!");
      Date lastAccess = (Date) accesses.get(remoteUser);
      if (lastAccess == null) {
        out.println("C'est votre première visite !");
      }
      else {
        out.println("Votre dernière visite date du " +
            accesses.get(remoteUser));
      }

      if (remoteUser.equals("PROFESSEUR FALKEN")) {
        out.println("Jouons-nous à un jeu ?");
      }

      accesses.put(remoteUser, new Date());
    }

    // ...Continuer le traitement de la requête...
  }
}
```

Cette servlet utilise un `Hashtable` pour stocker les dates de dernière visite de chaque utilisateur. À chaque requête, elle commence par accueillir la personne par son nom et lui indique la date de sa dernière visite. Ensuite, elle stocke la date de cette visite, pour l'utiliser la prochaine fois. Après cela, elle poursuit le traitement de la requête.

La requête

Nous avons vu comment les servlets obtiennent des informations sur le serveur et sur le client. Maintenant, il est temps de s'attaquer à ce qui est important : comment une servlet détermine ce que veut le client.

Paramètres de requête

Un nombre quelconque de paramètres peut être associé à chaque accès à une servlet. Typiquement, ces paramètres sont des paires nom/valeur qui donnent à la servlet toutes les informations additionnelles dont elle a besoin pour traiter la requête. Faites attention à ne pas confondre les paramètres de la requête et les paramètres d'initialisation, qui sont associés à la servlet elle-même.

Une servlet HTTP récupère ses paramètres de requête dans la chaîne d'interrogation (pour les requêtes GET) ou comme des données postées (pour les requêtes POST). Heureusement, chaque servlet retrouvent de la même façon ses paramètres, en utilisant `getParameter()` et `getParameterValues()` :

```
public String ServletRequest.getParameter(String name)
public String[] ServletRequest.getParameterValues(String name)
```

`getParameter()` retourne la valeur du paramètre nommé sous forme d'un `String` ou `null` si le paramètre n'a pas été spécifié[2]. Il est garanti que la valeur est dans sa forme normale décodée. S'il est possible qu'un paramètre puisse avoir plus d'une valeur, vous devez utiliser la méthode `getParameterValues()`. Cette méthode retourne toutes les valeurs du paramètre nommé sous forme d'un tableau de `String` ou `null` si le paramètre n'a pas été spécifié. Une seule valeur est retournée dans un tableau de longueur 1. Si vous appelez `getParameter()` sur un paramètre ayant plusieurs valeurs, la valeur retournée est la première valeur retournée par `getParameterValues()`.

Un mot d'avertissement : si l'information du paramètre vient dans des données POST, elle peut ne pas être disponible si ces données ont déjà été lues manuellement en utilisant les méthodes `getReader()` ou `getInputStream()` de `ServletRequest` (car les données POST ne peuvent être lues qu'une seule fois).

Les utilisations possibles des paramètres de requêtes sont illimitées. Ce sont des façons générales d'indiquer à une servlet ce qu'elle doit faire, comment elle doit le faire, ou les deux. Comme exemple simple, regardons comment une servlet dictionnaire pourrait utiliser `getParameter()` pour déterminer le mot qu'elle doit rechercher.

---

2. La méthode `getParameter()` est devenue obsolète dans Java Web Server 1.1 au profit de `getParameterValues()`. Cependant, après de nombreuses protestations du public, Sun a retiré `getParameter()` de la liste des désapprobations dans la version finale de l'API Servlet 2.0. Elle a été la première méthode Java à subir un tel revirement !

Un fichier HTML contenant le formulaire suivant permet à l'utilisateur d'entrer le mot à rechercher :

```
<FORM METHOD=GET ACTION="/servlet/Dictionary">
Mot à rechercher : <INPUT TYPE=TEXT NAME="word"><P>
<INPUT TYPE=SUBMIT><P>
</FORM>
```

Le code suivant retrouve le paramètre contenant le mot :

```
String word = req.getParameter("word");
String definition = getDefinition(word);
out.println(word + " : " + definition);
```

Ce code ne peut traiter qu'une seule valeur par paramètre. Certains paramètres ont plusieurs valeurs, par exemple avec l'emploi de <SELECT> :

```
<FORM METHOD=POST ACTION="/servlet/CarOrder">
Veuillez sélectionner les options souhaitées pour la Honda S2000 :<BR>
<SELECT NAME="features" MULTIPLE>
<OPTION VALUE="aero"> Déflecteur </OPTION>
<OPTION VALUE="cd"> Chargeur de CD </OPTION>
<OPTION VALUE="spoiler"> Becquet </OPTION>
</SELECT><BR>
<INPUT TYPE=SUBMIT VALUE="Ajouter au chariot">
</FORM>
```

Une servlet utilise la méthode getParameterValues() pour traiter ce formulaire :

```
String[] features = req.getParameterValues("features");
if (features != null) {
  for (int i = 0; i < features.length; i++) {
    cart.add(features[i]);
  }
}
```

En plus d'obtenir la valeurs des paramètres, une servlet peut accéder à leur nom par la méthode getParameterNames() :

```
public Enumeration ServletRequest.getParameterNames()
```

Cette méthode retourne le nom de tous les paramètres sous forme d'un Enumeration d'objets String ou d'un Enumeration vide si la servlet n'a pas de paramètre. Cette méthode est le plus souvent utilisée pour le débogage. L'ordre des noms ne correspond pas forcément à l'ordre donné dans le formulaire.

Enfin, une servlet peut retrouver la chaîne d'interrogation de la requête avec getQueryString() :

```
public String ServletRequest.getQueryString()
```

Cette méthode retourne la chaîne d'interrogation (l'information contenant les paramètres de GET encodés) de la requête ou null si la chaîne d'interrogation n'existe pas. Cette information de bas niveau est rarement utile pour traiter les données d'un formulaire. Elle est mieux adaptée au traitement d'une seule valeur sans nom, comme dans /servlet/Sqrt?576, où la chaîne d'interrogation retournée est "576".

L'*exemple 4-12* montre l'utilisation de ces méthodes avec une servlet qui imprime sa chaîne d'interrogation, puis imprime le nom et la valeur de tous ses paramètres.

Exemple 4-12. Examiner les paramètres

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ParameterSnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    out.println("Chaîne d'interrogation :");
    out.println(req.getQueryString());
    out.println();

    out.println("Paramètres de la requête :");
    Enumeration enum = req.getParameterNames();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      String values[] = req.getParameterValues(name);
      if (values != null) {
        for (int i = 0; i < values.length; i++) {
          out.println(name + " (" + i + ") : " + values[i]);
        }
      }
    }
  }
}
```

La sortie de cette servlet est montrée à la *figure 4-2*.

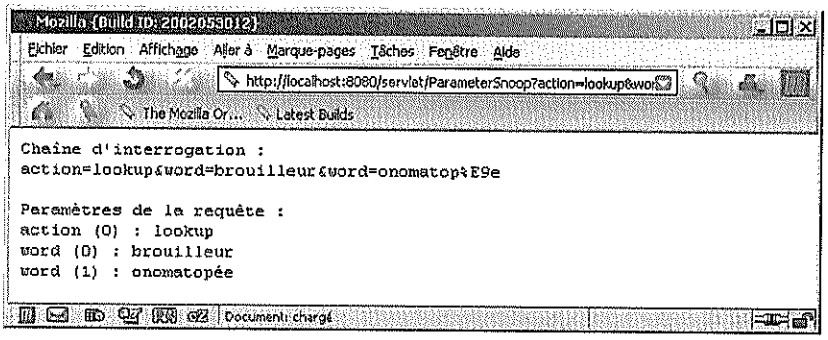


Figure 4-2. Les paramètres examinés

À partir de l'API Servlet 2.2, vous pouvez créer un formulaire POST avec une URL d'action qui contient une chaîne d'interrogation. Lorsque vous utilisez cette possibilité, les informations contenues dans les paramètres sont disponibles *via* les méthodes get-

Parameter(), la valeur des paramètres de la chaîne d'interrogation venant avant les valeurs de POST dans le cas de collisions de noms. Par exemple, si une requête possède la chaîne d'interrogation a=hokey et que les données POST sont a=pokey, la méthode req.getParameterValues("a") retourne le tableau { "hokey", "pokey" }.

Générer une clé de licence

Maintenant nous sommes prêts à écrire une servlet qui génère une clé de licence KeyedServerLock pour une machine et un numéro de port donnés. Une clé de cette servlet peut être utilisée pour déverrouiller la servlet KeyedServerLock. Comment cette servlet connaît-elle la machine et le numéro de port de la servlet qu'elle doit déverrouiller ? Avec les paramètres de requête, bien sûr. L'*exemple 4-13* montre le code correspondant.

Exemple 4-13. Déverrouiller KeyedServerLock

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class KeyedServerUnlock extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    PrintWriter out = res.getWriter();

    // Obtenir le machine et le port.
    String host = req.getParameter("host");
    String port = req.getParameter("port");

    // Si pas de machine, utiliser la machine courante.
    if (host == null) {
      host = req.getServerName();
    }

    // Convertir le port en un entier, si pas de port, utiliser le port
    // courant.
    int numericPort;
    try {
      numericPort = Integer.parseInt(port);
    }
    catch (NumberFormatException e) {
      numericPort = req.getServerPort();
    }

    // Générer et afficher la clé.
    // Toute KeyGenerationException est interceptée et affichée.
    try {
      long key = generateKey(host, numericPort);
```

Exemple 4-13. Déverrouiller KeyedServerLock (suite)

```
      out.println(host + ":" + numericPort + " a la clé " + key);
    }
    catch (KeyGenerationException e) {
      out.println("Impossible de générer une clé : " + e.getMessage());
    }
  }

  // Cette méthode contient l'algorithme utilisé pour faire correspondre une
  // clé avec un nom et un numéro de machine. Cette implémentation est très
  // faible et ne devrait pas être utilisée par des sites commerciaux.
  //
  // Lance une KeyGenerationException car toute autre chose plus spécifique
  // serait lié à l'algorithme choisi.
  //
  private long generateKey(String host, int port)
      throws KeyGenerationException {

    // La clé doit être un nombre sur 64 bits égal au non logique (~)
    // d'une adresse IP sur 32 bits concaténée au numéro de port sur 32 bits.

    byte hostIP[];
    try {
      hostIP = InetAddress.getByName(host).getAddress();

    }
    catch (UnknownHostException e) {
      throw new KeyGenerationException(e.getMessage());
    }

    // Obtenir l'adresse IP sur 32 bits.
    long servercode = 0;
    for (int i = 0; i < 4; i++) {
      servercode <<= 8;
      servercode |= hostIP[i];
    }

    // Concaténer le numéro de port sur 32 bits.
    servercode <<= 32;
    servercode |= port;

    // La clé est le non logique.
    return ~servercode;
  }
}

class KeyGenerationException extends Exception {

  public KeyGenerationException() {
```

Exemple 4-13. Déverrouiller KeyedServerLock (suite)

```
    super();
  }

  public KeyGenerationException(String msg) {
    super(msg);
  }
}
```

Vous pouvez utiliser la sortie de cette servlet pour affecter à KeyedServerLock une variable d'instance key spéciale :

```
<servlet>
  <servlet-name>
    ksl
  </servlet-name>
  <servlet-class>
    KeyedServerLock
  </servlet-class>
  <init-param>
    <param-name>
      key
    </param-name>
    <param-value>
      -9151314447111823249
    </param-value>
  </init-param>
</servlet>
```

N'oubliez pas de modifier la logique de generateKey() avant d'en faire toute utilisation réelle.

Informations de chemin

En plus des paramètres, une requête HTTP peut inclure ce que l'on appelle des *informations de chemin supplémentaire* ou un *chemin virtuel*. En général, ces informations de chemin supplémentaire servent à désigner un fichier sur le serveur que la servlet doit utiliser. L'information est codée dans l'URL d'une requête HTTP. En voici un exemple :

```
http://serveur:port/servlet/ViewFile/index.html
```

Cela invoque la servlet ViewFile, passant /index.html comme information de chemin supplémentaire. Une servlet peut avoir accès à cette information de chemin et elle peut aussi convertir la chaîne /index.html en un chemin réel vers le fichier *index.html*. Qu'est-ce que le chemin réel pour /index.html ? C'est le chemin complet du système de fichiers — ce que le serveur retournerait si le client demandait à accéder directement à /index.html. Cela serait probablement *racine_document/index.html*, mais, bien sûr, le serveur peut utiliser des alias spéciaux pour changer cela.

En dehors d'être spécifiée explicitement dans une URL, cette information de chemin supplémentaire peut aussi être codée dans le paramètre ACTION d'un formulaire HTML :

```
<FORM METHOD=GET ACTION="/servlet/Dictionary/dict/definitions.txt">
Mot à rechercher : <INPUT TYPE=TEXT NAME="word"><P>
<INPUT TYPE=SUBMIT><P>
</FORM>
```

Ce formulaire invoque la servlet `Dictionary` pour traiter les soumissions et passer à `Dictionary` l'information de chemin supplémentaire `/dict/definitions.txt`. La servlet sait ensuite quelles définitions rechercher en regardant dans le fichier *definitions.txt*, le même fichier que le client verrait s'il avait demandé `/dict/definitions.txt`, probablement `racine_serveur/webapps/ROOT/dict/definitions.txt` avec Tomcat.

Pourquoi des informations de chemin supplémentaire ?

Pourquoi HTTP a-t-il besoin d'un support spécial pour des informations de chemin supplémentaire ? N'est-il pas suffisant de passer à la servlet un paramètre de chemin ? La réponse est oui. Les servlets n'ont pas besoin de ce support spécial, mais les programmes CGI si.

Un programme CGI ne peut pas interagir avec son serveur pendant son exécution. Il n'a donc aucun moyen de recevoir un paramètre de chemin, ni de demander au serveur de lui faire correspondre un vrai emplacement dans le système de fichiers. Le serveur doit en quelque sorte traduire le chemin avant d'invoquer le programme CGI. C'est pourquoi le support spécial des « informations de chemin supplémentaire » existe. Les serveurs savent prétraduire ce chemin et envoyer la traduction au programme CGI dans une variable d'environnement. C'est une façon assez élégante de contourner un problème de CGI.

Bien sûr, ce n'est pas parce que les servlets n'ont pas vraiment besoin de ce traitement spécial qu'elles ne doivent pas l'utiliser. C'est une façon simple et pratique d'attacher un chemin à une requête.

Obtenir l'information de chemin

Une servlet utilise la méthode `getPathInfo()` pour obtenir l'information de chemin supplémentaire :

```
public String HttpServletRequest.getPathInfo()
```

Cette méthode retourne l'information de chemin supplémentaire associée à la requête (URL décodée si nécessaire) ou `null` si aucune n'était donnée. Un exemple de chemin est `/dict/definitions.txt`. L'information de chemin, en elle-même, n'est que marginalement utile. Habituellement, une servlet a besoin de connaître le vrai emplacement, sur le système de fichiers, du fichier donné dans l'information de chemin. C'est là que la méthode `getPathTranslated()` apparaît :

```
public String HttpServletRequest.getPathTranslated()
```

Cette méthode retourne l'information de chemin supplémentaire traduite en chemin réel sur le système de fichiers (URL décodée si nécessaire) ou `null` si l'information de chemin n'existait pas. La méthode retourne aussi `null` si le chemin ne peut être traduit en un chemin de fichier raisonnable, par exemple lorsque l'application web s'exécute à

partir d'un fichier WAR, d'un système de fichier distant non disponible localement ou d'une base de données. Le chemin retourné ne pointe pas nécessairement sur un répertoire ou un fichier existant. Un exemple de chemin traduit est `C:\tomcat\webapps\ROOT\dict\definitions.txt`.

L'*exemple 4-14* montre une servlet qui utilise ces deux méthodes pour afficher l'information de chemin supplémentaire qu'elle reçoit et la traduction résultante en un chemin réel.

Exemple 4-14. Montrer où conduit le chemin

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FileLocation extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    if (req.getPathInfo() != null) {
      out.println("Le fichier \"" + req.getPathInfo() + "\"");
      out.println("est stocké dans \"" + req.getPathTranslated() + "\"");
    }
    else {
      out.println("L'information de chemin est nulle, "
                  + "aucun fichier à retrouver");
    }
  }
}
```

Voici un exemple de sortie de cette servlet :

```
Le fichier "/index.html"
est stocké dans "/usr/local/tomcat/webapps/ROOT/index.html"
```

Traductions de chemin ad hoc

Parfois une servlet a besoin de traduire un chemin qui n'a pas été passé dans l'information de chemin supplémentaire. Pour cela, vous pouvez utiliser la méthode `getRealPath()` :

```
public String ServletContext.getRealPath(String path)
```

Cette méthode retourne le chemin réel pour tout *chemin virtuel* donné ou `null` si la traduction ne peut être effectuée. Si le chemin passé est `/`, la méthode retourne la racine des documents (l'endroit où les documents sont stockés) pour le serveur. Si le chemin passé est `getPathInfo()`, la méthode retourne le même chemin réel qu'aurait retourné `getPathTranslated()`. Cette méthode peut être utilisée aussi bien par des servlets génériques que par des servlets HTTP. Il n'y a pas d'équivalent CGI.

Obtenir le chemin de contexte

Vous avez appris au chapitre 2, *Introduction aux servlets HTTP*, que les applications web sont liées à des préfixes d'URI sur le serveur. Une servlet peut déterminer le préfixe d'URI du contexte dans lequel elle s'exécute à l'aide de la méthode `getContextPath()` fournie par `HttpServletRequest` :

```
public String HttpServletRequest.getContextPath()
```

Cette méthode retourne un `String` représentant le préfixe d'URI du contexte traitant la requête. La valeur commence par /, n'a pas de / final et, pour le contexte par défaut, est vide. Par exemple, pour la requête */catalogue/livres/servlet/Achat*, la méthode `getContextPath()` retourne `/catalogue/livres`.

Vous pouvez utiliser cette méthode pour qu'il soit plus facile de développer des servlets qui fonctionnent quel que soit le chemin de contexte auquel elles sont associées. Par exemple, pour générer lien vers la page d'accueil d'un contexte, ne figez pas le chemin de contexte dans le code mais utilisez le code suivant :

```
out.println("<a href=\"" + req.getContextPath() +
    "/index.html\">Accueil</a>");
```

Obtenir des types MIME

Une fois que la servlet a obtenu le chemin d'un fichier, elle a souvent besoin de découvrir son type. C'est à cela que sert la méthode `getMimeType()` :

```
public String ServletContext.getMimeType(String file)
```

Cette méthode retourne le type MIME du fichier indiqué, d'après son extension, ou `null` s'il n'est pas connu. Les types MIME communs sont `text/html`, `text/plain`, `image/gif` et `image/jpeg`. Le fragment de code suivant détermine le type MIME de l'information de chemin supplémentaire :

```
String type = getServletContext().getMimeType(req.getPathTranslated())
```

Les serveurs connaissent en général un ensemble de base des correspondances extension de fichier/type MIME. Il peut être modifié ou étendu par des entrées dans le descripteur de déploiement *web.xml*, donnant ainsi à chaque contexte un comportement configurable, comme à l'*exemple 4-15*.

Exemple 4-15. Tout le monde aime un MIME

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <!-- ..... -->
    <mime-mapping>
        <extension>
            java
        </extension>
```

Exemple 4-15. Tout le monde aime un MIME (suite)

```
        <mime-type>
            text/plain
        </mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>
            cpp
        </extension>
        <mime-type>
            text/plain
        </mime-type>
    </mime-mapping>
</web-app>
```

Servir des fichiers

Tomcat Server utilise lui-même des servlets pour traiter chaque requête. En plus de constituer un parfait exemple des possibilité des servlets, cela apporte au serveur une conception modulaire qui permet un remplacement global de certains aspects de ses fonctionnalités. Par exemple, tous les fichiers sont servis par la servlet `org.apache.tomcat.core.DefaultServlet` chargée de traiter l'alias / (elle est donc le gestionnaire par défaut des requêtes). Mais il n'est pas question de dire que `DefaultServlet` ne peut pas être remplacée. Elle peut l'être, en indiquant que le motif d'URL / doit utiliser une autre servlet. Qui plus est, il n'est pas si difficile que cela d'écrire une version de remplacement de `DefaultServlet` en utilisant les méthodes que nous venons de voir.

L'*exemple 4-16* montre une servlet `ViewFile` qui utilise les méthodes `getPathTranslated()` et `getMimeType()` pour retourner tout fichier indiqué par l'information de chemin supplémentaire.

Exemple 4-16. Retourner dynamiquement des fichiers statiques

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class ViewFile extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    // Utiliser un ServletOutputStream car l'information passée peut être binaire.
    ServletOutputStream out = res.getOutputStream();

    // Récupérer le fichier à visualiser.
    String file = req.getPathTranslated();
```

Exemple 4-16. Retourner dynamiquement des fichiers statiques (suite)

```
    // Pas de fichier, rien à montrer.
    if (file == null) {
      out.println("Aucun fichier à visualiser");
      return;
    }

    // Récupérer et définir le type de ce fichier.
    String contentType = getServletContext().getMimeType(file);
    res.setContentType(contentType);

    // Retourner le fichier.
    try {
      ServletUtils.returnFile(file, out);
    }
    catch (FileNotFoundException e) {
      out.println("Fichier non trouvé");
    }
    catch (IOException e) {
      out.println("Problème d'envoi du fichier : " + e.getMessage());
    }
  }
}
```

Cette servlet commence par utiliser `getPathTranslated()` pour obtenir le nom du fichier qu'elle doit afficher. Ensuite, elle utilise `getMimeType()` pour déterminer le type du contenu de ce fichier et lui faire correspondre le type du contenu de la réponse. Enfin, elle retourne le fichier en utilisant la méthode `returnFile()` fournie par la classe utilitaire `com.oreilly.servlet.ServletUtils` :

```
  // Envoyer le contenu du fichier sur le flot de sortie.
  public static void returnFile(String filename, OutputStream out)
                                throws FileNotFoundException, IOException {
    // Un FileInputStream manipule des octets.
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(filename);
      byte[] buf = new byte[4 * 1024];  // Tampon de 4K.
      int bytesRead;
      while ((bytesRead = fis.read(buf)) != -1) {
        out.write(buf, 0, bytesRead);
      }
    }
    finally {
      if (fis != null) fis.close();
    }
  }
```

Le traitement des erreurs de la servlet est basique, il consiste à retourner une page décrivant l'erreur. C'est acceptable pour notre exemple simple (et en fait bien plus consé-

quent que ce dont semblent capables plusieurs programmes), mais nous verrons une meilleure façon d'utiliser les codes d'état dans le prochain chapitre. Cette servlet peut être utilisée directement avec une URL comme la suivante :

```
http://serveur:port/servlet/ViewFile/index.html
```

Ou, si vous affectez cette servlet au traitement du motif d'URL par défaut :

```
<servlet>
    <servlet-name>
        vf
    </servlet-name>
    <servlet-class>
        ViewFile
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>
        vf
    </servlet-name>
    <url-pattern>
        /
    </url-pattern>
</servlet-mapping>
```

`ViewFile` est alors invoquée automatiquement même avec une URL du type suivant :

```
http://serveur:port/index.html
```

Notez quand même que cette servlet est un cas d'école et qu'elle n'a pas autant de fonctionnalités que `DefaultServlet`.

Lire à partir d'une ressource abstraite

La méthode `getPathTranslated()` possède quelques limitations malheureuses. Tout d'abord, elle ne fonctionne pas avec le contenu provenant de fichiers WAR car il n'y a pas de fichier avec un accès direct. Ensuite, elle ne fonctionne pas dans un environnement distribué avec équilibrage de charge où il peut y avoir un fichier direct mais pas sur le serveur exécutant la servlet. Pour contourner ces limitations, l'API Servlet 2.1 a introduit une technique pour l'abstraction des ressources permettant aux servlets d'accéder à une ressource sans savoir où elle réside. Une servlet obtient un accès à une ressource abstraite *via* la méthode `getResource()` :

```
public URL ServletContext.getResource(String uripath)
```

Cette méthode retourne une URL pouvant servir à interroger la ressource spécifiée et à lire son contenu. C'est le serveur web qui détermine la manière de relier le paramètre de chemin d'URI et la ressource réelle (fichier, entrée WAR, entrée de base de données, etc.). Les deux restrictions sont que le chemin doit être absolu (commencer par une barre oblique) et que l'URI ne doit pas être une ressource active comme une autre servlet ou un programme CGI. La méthode `getResource()` sert uniquement au support de la lecture de contenu statique (pas de contenu dynamique, ni d'écriture de contenu).

Quand on se sert de l'objet de contexte pour demander une ressource, il est important de se rappeler de ne *pas* inclure le chemin de contexte dans la requête. Après tout, le

contexte connaît son propre chemin et en ne spécifiant pas le chemin dans le code, vous garantissez un déplacement possible de l'application vers un préfixe de chemin différent sans nécessiter une recompilation. Le code suivant récupère et affiche le fichier */includes/header.html* du contexte courant :

```
URL url = getServletContext().getResource("/includes/header.html");
if (url != null) {
  ServletUtils.returnURL(url, out);
}
```

Le fichier *header.html* peut se trouver dans un fichier d'archive sur une machine serveur autre que celle hébergeant la servlet, mais cela n'a pas d'importance. Le code appelle la méthode utilitaire `returnURL()` de la classe `com.oreilly.servlet.ServletUtils` :

```
// Envoyer le contenu de l'URL dans un OutputStream.
public static void returnURL(URL url, OutputStream out)
    throws IOException {
  InputStream in = url.openStream();
  byte[] buf = new byte[4 * 1024];  // Tampon de 4K.
  int bytesRead;
  while ((bytesRead = in.read(buf)) != -1) {
    out.write(buf, 0, bytesRead);
  }
}

// Envoyer le contenu de l'URL dans un PrintWriter.
public static void returnURL(URL url, PrintWriter out)
    throws IOException {
  // Déterminer le codage du contenu de l'URL.
  URLConnection con = url.openConnection();
  con.connect();
  String encoding = con.getContentEncoding();

  // Construire un Reader adapté à ce codage.
  BufferedReader in = null;
  if (encoding == null) {
    in = new BufferedReader(
         new InputStreamReader(url.openStream()));
  }
  else {
    in = new BufferedReader(
         new InputStreamReader(url.openStream(), encoding));
  }
  char[] buf = new char[4 * 1024];  // Tampon de 4K caractères.
  int charsRead;
  while ((charsRead = in.read(buf)) != -1) {
    out.write(buf, 0, charsRead);
  }
}
```

Comme vous pouvez le voir dans la deuxième méthode `returnURL()` du code précédent, il est possible d'utiliser l'objet d'URL pour rechercher les attributs d'une ressource abstraite. Cette fonctionnalité n'est pas supportée par tous les serveurs ni par tous les envi-

ronnement d'exécution Java. Voici le code qui examine la page d'accueil du contexte courante :

```
URL url = getServletContext().getResource("/index.html"); // Page d'accueil
                                                          // du contexte.
URLConnection con = url.openConnection();
con.connect();
int contentLength = con.getContentLength();     // Support non garanti.
String contentType = con.getContentType();      // Support non garanti.
long expiration = con.getExpiration();          // Support non garanti.
long lastModified = con.getLastModified();      // Support non garanti.
// etc...
```

Souvenez-vous que le contenu servi pour le chemin d'URI/ est entièrement déterminé par le serveur. Pour accéder aux ressources à partir d'un autre contexte, vous pouvez employer `getContext()` :

```
public ServletContext ServletContext.getContext(String uripath)
```

Cette méthode retourne une référence au `ServletContext` de l'URI donné, sujet à de possibles contraintes de sécurité imposées par le serveur. Voici comment obtenir le contexte par défaut :

```
getServletContext().getContext("/");
```

Et voici comment obtenir une référence à la page d'accueil du serveur web :

```
getServletContext().getContext("/").getResource("/index.html");
```

Sachez que `getResource()` ne connaît pas forcément la liste des fichiers de bienvenue. `getResource("/")` peut donc retourner un contenu inutilisable.

Il existe aussi une méthode utilitaire, `getResourceAsStream()`, qui lit des ressources sous forme d'un flot :

```
public InputStream ServletContext.getResourceAsStream(String uripath)
```

Elle se comporte essentiellement comme `getResource().openStream()`. Pour des raisons de compatibilité et pour qu'il soit plus facile aux programmeurs CGI de passer aux servlets, l'API Servlet inclut toujours des méthodes d'accès aux fichiers, comme `getPathTranslated()`. Rappelez-vous simplement que chaque fois que vous accéder à une ressource à l'aide d'un objet `File`, vous vous liez à une machine particulière.

Servir des ressources

À l'aide des ressources abstraites, nous pouvons améliorer notre version de `ViewFile` afin qu'elle fonctionne avec du contenu provenant d'un fichier WAR ou résidant sur une machine différente de celle qui exécute la servlet. La servlet `UnsafeViewResource` est montrée à l'*exemple 4-17*. Elle est dite « non sûre » car elle n'apporte aucune protection des ressources et sert aveuglément les fichiers de *WEB-INF* et les sources *.jsp*.

Exemple 4-17. Servir une ressource abstraite, de façon non sûre

```
import java.io.*;
import java.net.*;
import java.util.*;
```

Exemple 4-17. Servir une ressource abstraite, de façon non sûre (suite)

```
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class UnsafeViewResource extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // Utiliser un ServletOutputStream car les informations peuvent
    // être binaires.
    ServletOutputStream out = res.getOutputStream();
    res.setContentType("text/plain");  // Valeur par défaut.

    // Obtenir la ressource à afficher.
    String file = req.getPathInfo();
    if (file == null) {
      out.println("L'information de chemin supplémentaire est nulle ; " +
                  "il faut une ressource à afficher");
      return;
    }

    // Convertir la ressource en une URL.
    // ATTENTION : cela permet d'accéder aux fichiers du répertoire WEB-INF
    // et aux sources .jsp.
    URL url = getServletContext().getResource(file);
    if (url == null) {  // Certains serveurs retournent null si non trouvé.
      out.println("Ressource " + file + " non trouvée");
      return;
    }

    // Se connecter à la ressource.
    URLConnection con = null;
    try {
      con = url.openConnection();
      con.connect();
    } catch (IOException e) {
      out.println("Impossible de lire la ressource " + file + " : " +
        e.getMessage());
      return;
    }

    // Obtenir et définir le type de la ressource.
    String contentType = con.getContentType();
    res.setContentType(contentType);

    // Retourner la ressource.
    // ATTENTION : cela renvoie les fichiers du répertoire WEB-INF et
    // les sources .jsp.
```

Exemple 4-17. Servir une ressource abstraite, de façon non sûre (suite)

```
    try {
      ServletUtils.returnURL(url, out);
    } catch (IOException e) {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
              "Problème avec l'envoi de la ressource : " + e.getMessage());
    }
  }
}
```

Cette servlet affiche uniquement les fichiers de son propre contexte. Tout fichier en dehors de ce contexte est inaccessible à partir de la méthode `getServletContext().getResource()`. Cette servlet ne protège absolument pas les ressources. Les fichiers qui se trouvent dans le répertoire *WEB-INF* et les sources *.jsp* peuvent donc être envoyés directement. L'*exemple 4-18* montre une version plus sûre de la classe. Elle utilise la méthode `ServletUtils.getResource()` de `com.oreilly.servlet`.

Exemple 4-18. Servir une ressource abstraite, de façon sûre

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class ViewResource extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // Utiliser un ServletOutputStream car les informations peuvent
    // être binaires.
    ServletOutputStream out = res.getOutputStream();
    res.setContentType("text/plain");  // Valeur par défaut.

    // Obtenir la ressource à afficher.
    URL url = null;
    try {
      url = ServletUtils.getResource(getServletContext(), req.getPathInfo());
    } catch (IOException e) {
      res.sendError(
        res.SC_NOT_FOUND,
        "L'information de chemin supplémentaire doit désigner une " +
        "ressource valide : " + e.getMessage());
      return;
    }

    // Se connecter à la ressource.
    URLConnection con = url.openConnection();
    con.connect();
```

Exemple 4-18. Servir une ressource abstraite, de façon sûre (suite)

```
    // Obtenir et définir le type de la ressource.
    String contentType = con.getContentType();
    res.setContentType(contentType);

    // Retourner la ressource.
    try {
      ServletUtils.returnURL(url, out);
    } catch (IOException e) {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
              "Problème d'envoi de la ressource : " + e.getMessage());
    }
  }
}
```

La méthode ServletUtils.getResource() enveloppe la méthode context.getResource() et ajoute trois contrôles de sécurité : les ressources ne sont pas servies si elles contiennent des doubles points, si elles se terminent par une barre oblique, un point ou par *.jsp*, ou si elles commencent par *WEB-INF* ou *META-INF*. Voici le code :

```
public static URL getResource(ServletContext context, String resource)
                                        throws IOException {
  // Raccourci si la ressource est null.
  if (resource == null) {
    throw new FileNotFoundException(
      "La ressource demandée est nulle (null a été passé)");
  }

  if (resource.endsWith("/") ||
      resource.endsWith("\\") ||
      resource.endsWith(".")) {
    throw new MalformedURLException(
      "Le chemin ne doit pas se terminer par une barre oblique " +
      "ou un point");
  }

  if (resource.indexOf("..") != -1) {
    throw new MalformedURLException(
      "Le chemin ne doit pas contenir des doubles points");
  }

  String upperResource = resource.toUpperCase();
  if (upperResource.startsWith("/WEB-INF") ||
      upperResource.startsWith("/META-INF")) {
    throw new MalformedURLException(
      "Le chemin ne doit pas commencer par /WEB-INF ou /META-INF");
  }

  if (upperResource.endsWith(".JSP")) {
    throw new MalformedURLException(
```

```
        "Le chemin ne doit pas se terminer par .jsp");
    }

    // Convertir la ressource en une URL.
    URL url = context.getResource(resource);
    if (url == null) {
      throw new FileNotFoundException(
        "La ressource demandée est nulle (" + resource + ")");
    }

    return url;
  }
```

Servir des ressources en téléchargement

La servlet `ViewResource` a une utilisation plus large que pour l'apprentissage. Si la servlet `ViewResource` fige dans le code le `Content-Type` de la réponse à `application/octet-stream`, elle peut servir de système de téléchargement générique. Quand ils reçoivent un contenu de type `application/octet-stream`, la plupart des navigateurs affichent à l'utilisateur une fenêtre de dialogue lui demandant de sauvegarder le contenu. L'exception est Microsoft Internet Explorer qui, s'il reconnaît empiriquement le type du contenu, ignore le type de contenu indiqué par le serveur et l'affiche normalement. Cette « fonctionnalité » empêche le téléchargement des fichiers de type GIF, JPEG, HTML ou autres dans un navigateur Internet Explorer.

Déterminer ce qui a été demandé

Une servlet peut utiliser plusieurs méthodes pour déterminer exactement quel fichier ou quelle servlet le client a demandé. Après tout, seule la plus prétentieuse des servlets peut supposer qu'elle est toujours la cible directe d'une requête. Une servlet peut n'être rien de plus qu'un simple intermédiaire d'un autre contenu.

Aucune méthode ne retourne directement l'URL (*Uniform Resource Locator*) originale utilisée par le client pour effectuer une requête. Cependant, la classe `javax.servlet.http.HttpUtils` fournit une méthode `getRequestURL()` qui fait à peu près cela :[3]

```
public static StringBuffer HttpUtils.getRequestURL(HttpServletRequest req)
```

Cette méthode reconstruit l'URL de la requête en fonction des informations disponibles dans l'objet `HttpServletRequest`. Elle retourne un `StringBuffer` qui inclut le schéma (comme HTTP), le nom du serveur, le port du serveur et l'information de chemin supplémentaire. L'URL reconstruite doit être presque identique à l'URL utilisée par le client. Les différences entre l'URL originale et l'URL reconstruite doivent être mineures (c'est-à-dire, un espace codé `%20` par le client pourra être codé + par le serveur). Puisque

3. Pourquoi n'y a-t-il pas une méthode qui retourne directement l'URL originale montrée dans le navigateur ? Parce que le navigateur n'envoie jamais l'URL complète. Le numéro de port, par exemple, est utilisé par le client pour effectuer sa connexion HTTP, mais il n'est pas inclus dans la requête envoyée au serveur Web qui répond sur ce numéro de port.

cette méthode retourne un `StringBuffer`, l'URL de la requête peut être modifiée efficacement (par exemple, en ajoutant les paramètres de requête). Cette méthode est souvent utilisée pour créer des messages de redirection et rapporter les erreurs.

Cependant, la plupart du temps, la servlet n'a pas réellement besoin de l'URL de la requête. Elle a simplement besoin de l'URI de la requête, retourné par `getRequestURI()` :

```
public String HttpServletRequest.getRequestURI()
```

Cette méthode retourne l'URI (*Universal Resource Identifier*) de la requête. Pour des servlets HTTP normales, l'URI d'une requête peut être vu comme une URL de laquelle ont été supprimés le schéma, la machine, le port et la chaîne d'interrogation, mais incluant toute information de chemin supplémentaire. Autrement dit, il s'agit du chemin de contexte, plus le chemin de servlet, plus l'information de chemin.[4] Le *tableau 4-2* montre les URI pour différentes URL.

Tableau 4-2. URL et leur URI

URL de requête	URI correspondant
http://serveur:port/servlet/NomClasse	*/servlet/NomClasse*
http://serveur:port/servlet/NomEnregistre	*/servlet/NomEnregistre*
http://serveur:port/servlet/NomClasse?var=val	*/servlet/NomClasse*
http://serveur:port/servlet/NomClasse/infochemin	*/servlet/NomClasse/infochemin*
http://serveur:port/servlet/NomClasse/infochemin?var=val	*/servlet/NomClasse/infochemin*
http://serveur:port/servlet/NomClasse/info%20chemin[a]	*/servlet/NomClasse/info%20chemin*
http://serveur:port/alias.html (alias d'une servlet)	*/alias.html*
http://serveur:port/chemin/contexte/servlet/Classname	*/chemin/contexte/servlet/NomClasse*

a. %20 est la version codée d'un espace.

Dans certains cas, une servlet a seulement besoin de connaître le nom sous lequel elle a été invoquée. Vous pouvez retrouver cette information avec la méthode `getServletPath()` :

```
public String HttpServletRequest.getServletPath()
```

Cette méthode retourne la partie de l'URI qui fait référence à la servlet invoquée (URL décodée si nécessaire) ou `null` si l'URI ne pointe pas directement sur une servlet. Le chemin de la servlet n'inclut pas d'information de chemin supplémentaire ou de chemin de contexte. Le *tableau 4-3* montre les noms de servlet pour plusieurs URL de requête.

Voici une astuce vous permettant de vous souvenir de l'information de chemin :

```
decoded(getRequestURI) ==
            decoded(getContextPath) + getServletPath + getPathInfo
```

4. Techniquement, ce que l'on désigne par URI de requête devrait plus formellement être appelé un *chemin d'URL de requête*. Cela parce qu'un URI est, dans un sens plus précis, un identificateur de ressource général. Une URL est un type d'URI ; un URN (*Uniform Resource Name*) en est un autre. Pour plus d'informations sur les URI, URL et URN, reportez-vous à la RFC 1630 à *http://www.ietf.org/rfc/rfc1630.txt*.

Tableau 4-3. URL et leur chemin de servlet

URL de requête	Chemin de servlet
http://serveur:port/servlet/NomClasse	*/servlet/NomClasse*
http://serveur:port/servlet/NomEnregistre	*/servlet/NomEnregistre*
http://serveur:port/servlet/NomClasse?var=val	*/servlet/NomClasse*
http://serveur:port/servlet/NomClasse/infochemin	*/servlet/NomClasse*
http://serveur:port/servlet/NomClasse/infochemin?var=val	*/servlet/NomClasse*
http://serveur:port/servlet/NomClasse/infos%20chemin	*/servlet/NomClasse*
http://serveur:port/alias.html (alias d'une servlet)	*/alias.html*
http://serveur:port/chemin/contexte/servlet/NomClasse	*/servlet/NomClasse*

Comment la demande a-t-elle été faite ?

En dehors de savoir ce qui a été demandé, une servlet dispose de plusieurs moyens pour obtenir les détails sur la façon dont cette demande s'est faite. La méthode `getScheme()` retourne le schéma utilisé pour effectuer la requête :

```
public String ServletRequest.getScheme()
```

Des exemples de schéma sont `http`, `https` et `ftp`, ainsi que les schémas plus récents spécifiques à Java, `jdbc` et `rmi`. Il n'existe pas d'équivalent CGI direct (même si certaines implémentations CGI ont une variable `SERVER_URL` qui inclut le schéma). Pour les servlets HTTP, cette méthode indique si la requête a été faite au-dessus d'une connexion sécurisée à l'aide de SSL (*Secure Sockets Layer*), comme le signale le schéma `https`, ou si elle était non sûre, comme l'indique le schéma `http`.

La méthode `getProtocol()` retourne le protocole et le numéro de version utilisés pour effectuer la requête :

```
public String ServletRequest.getProtocol()
```

Le protocole et le numéro de version sont séparés par une barre oblique. La méthode retourne `null` si le protocole ne peut être déterminé. Pour les servlets HTTP, le protocole est généralement `HTTP/1.0` ou `HTTP/1.1`. Les servlets HTTP utilisent la version du protocole pour déterminer si le client peut utiliser les nouvelles fonctionnalités de la version 1.1 de HTTP.

Pour déterminer la méthode utilisée par une requête, une servlet utilise `getMethod()` :

```
public String HttpServletRequest.getMethod()
```

Cette méthode retourne la méthode HTTP utilisée pour effectuer la requête. Parmi ces méthodes, on trouve `GET`, `POST` et `HEAD`. La méthode `service()` de l'implémentation de `HttpServlet` utilise cette méthode pour distribuer les requêtes.

En-têtes de requête

Aux requêtes et aux réponses HTTP peuvent être associés des *en-têtes* HTTP. Ces en-têtes fournissent des informations supplémentaires sur la requête ou sur la réponse. Le protocole HTTP version 1.0 définit des dizaines d'en-têtes possibles ; le protocole HTTP

version 1.1 en définit encore plus. Une description de ces en-têtes sort du cadre de ce livre ; nous discutons uniquement des en-têtes les plus souvent utilisés par les servlets. Pour une liste complète des en-têtes HTTP et de leur utilisation, nous vous recommandons le livre *Web Client Programming* de Clinton Wong (O'Reilly & Associates) ou *Webmaster in a Nutshell* de Stephen Spainhour et Robert Eckstein (Éditions O'Reilly).

Une servlet a rarement besoin de lire les en-têtes HTTP accompagnant une requête. Plusieurs de ces en-têtes sont traités par le serveur lui-même. Examinons, par exemple, comment un serveur restreint l'accès à ses documents. Le serveur utilise les en-têtes HTTP et les servlets n'ont pas besoin de connaître les détails. Quand le serveur reçoit une requête pour une page à accès restreint, il vérifie que cette requête inclut un en-tête `Authorization` qui contient un nom d'utilisateur et un mot de passe valides. Si ce n'est pas le cas, le serveur retourne lui-même une réponse contenant un en-tête `WWW-Authenticate`, pour indiquer au navigateur que l'accès à la ressource lui a été refusé. Lorsque le client envoie une requête contenant un en-tête valide, le serveur autorise l'accès et donne à la servlet invoquée l'accès au nom de l'utilisateur *via* la méthode `getRemoteUser()`.

D'autres en-têtes sont utilisés par les servlets, mais indirectement. Un bon exemple est `Last-Modified` et `If-Last-Modified` dont on a parlés au chapitre 3. Le serveur voit l'en-tête `If-Last-Modified` et appelle la méthode `getLastModified()` de la servlet pour déterminer comment continuer.

Il y a aussi quelques en-têtes HTTP qu'une servlet peut vouloir lire à l'occasion. Le *tableau 4-4* en donne la liste.

Tableau 4-4. En-têtes de requête HTTP utiles

En-tête	Utilisation
`Accept`	Indique le type de média (MIME) que le client préfère recevoir, séparé par des virgules. Certains navigateurs anciens envoient un en-tête séparé pour chaque type de média. Chaque type de média est divisé en un type et un sous-type sous la forme *type/sous-type*. Un astérisque étoile (*) est autorisé pour le sous-type (*type/**) ou pour les deux (*/*). Par exemple : `Accept: image/gif, image/jpeg, text/*, */*` Une servlet peut utiliser cet en-tête pour déterminer le type de contenu à retourner. Si cet en-tête n'est pas donné dans la requête, la servlet peut supposer que le client accepte tous les types de média.
`Accept-Language`	Spécifie la ou les langues que le client préfère recevoir. Le format utilisé est l'abréviation de langues standard de l'ISO-639 avec un code de pays ISO-3166 optionnel. Par exemple : `Accept-Language: fr, en, es, de, ja, zh-TW` Cela indique que le client lit le français, l'anglais, l'espagnol, l'allemand, le japonais et le chinois adapté à Taiwan. Par convention, les langues sont données dans l'ordre de préférence. Voir le chapitre 13, *Internationalisation*, pour plus d'informations sur la négociation de la langue.

Tableau 4-4. En-têtes de requête HTTP utiles (suite)

En-tête	Utilisation

`User-Agent`	Donne des informations sur le logiciel du client. Le format de la chaîne de caractères retournée est plutôt libre, mais il inclut souvent le nom et la version du navigateur ainsi que des informations sur la machine sur laquelle il s'exécute. Netscape 4.7 sur une machine SGI Indy avec IRIX 6.2 indique :
	`User-Agent: Mozilla/4.7 [en] (X11; U; IRIX 6.2 IP22)`
	Microsoft Internet Explorer 4.0 sur une machine Windows 95 indique :
	`User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)`
	Une servlet peut utiliser cet en-tête pour conserver des statistiques ou pour personnaliser ses réponses en fonction du navigateur.

`Referer`	Donne l'URL du document qui fait référence à l'URL demandée (c'est-à-dire, le document qui contient le lien suivi par le client pour accéder à ce document).[a] Par exemple : `Referer: http://developer.java.sun.com/index.html` Une servlet peut utiliser cet en-tête pour conserver des statistiques ou, s'il y a eu une erreur dans la requête, pour garder une trace des documents contenant des erreurs.
`Authorization`	Fournit l'autorisation du client pour accéder à l'URI, dont le nom d'utilisateur et le mot de passe codés en Base64. Les servlets peuvent l'utiliser pour personnaliser les autorisations, comme nous le verrons au chapitre 8, *Sécurité*.

a. Un dictionnaire anglais donnerait le mot `Referrer`. Cependant, nous devons nous conformer à la spécification qui l'écrit `Referer`.

Accès aux valeurs des en-têtes

On accède à la valeur des en-têtes HTTP *via* l'objet `HttpServletRequest`. Une valeur d'en-tête peut être trouvée sous forme d'un `String`, d'un `long` (représentant un `Date`) ou d'un `int`, en utilisant `getHeader()`, `getDateHeader()` et `getIntHeader()`, respectivement :

```
public String HttpServletRequest.getHeader(String name)
public long HttpServletRequest.getDateHeader(String name)
public int HttpServletRequest.getIntHeader(String name)
```

`getHeader()` retourne la valeur de l'en-tête indiqué sous forme d'un `String` ou `null` si l'en-tête n'a pas été envoyé dans la requête. Le nom est sensible à la casse, comme c'est le cas pour toutes ces méthodes. Tous les types d'en-têtes peuvent être retrouvés avec cette méthode.

`getDateHeader()` retourne la valeur de l'en-tête indiqué sous forme d'un `long` (représentant un `Date`) qui indique le nombre de milli-secondes depuis l'origine ou `-1` si l'en-tête n'a pas été envoyé dans la requête. Cette méthode lance une exception `IllegalArgumentException` lorsque la valeur de l'en-tête ne peut pas être convertie en un `Date`. La méthode est utile pour traiter les en-têtes comme `Last-Modified` et `If-Modified`.

`getIntHeader()` retourne la valeur de l'en-tête indiqué sous forme d'un `int` ou `-1` si l'en-tête n'a pas été envoyé dans la requête. Cette méthode lance une exception `NumberFormatException` lorsque la valeur de l'en-tête ne peut pas être convertie en un `int`.

Une servlet peut aussi obtenir le nom de tous les en-têtes en utilisant `getHeaderNames()` :

```
public Enumeration HttpServletRequest.getHeaderNames()
```

Cette méthode retourne le nom de tous les en-têtes sous forme d'un `Enumeration` d'objets `String`. Elle retourne un `Enumeration` vide s'il n'y a pas d'en-tête. L'API Servlet permet aux implémentations des conteneurs de servlets d'interdire cette forme d'accès aux en-têtes, auquel cas la méthode retourne `null`.

Certaines en-têtes, comme `Accept` et `Accept-Language`, supportent les valeurs multiples. Normalement, ces valeurs sont passées dans un seul en-tête, séparées par des espaces, mais certains navigateurs préfèrent envoyer les valeurs multiples *via* autant d'en-têtes :

```
Accept-Language: fr
Accept-Language: en
Accept-Language: ja
```

Pour lire un en-tête ayant de multiples valeurs, les servlets utilisent la méthode `getHeaders()` :

```
public Enumeration HttpServletRequest.getHeaders(String name)
```

Cette méthode retourne toutes les valeurs d'un en-tête donné sous forme d'un `Enumeration` d'objets `String` ou un `Enumeration` vide si l'en-tête n'était pas dans la requête. Si le conteneur de servlets n'autorise pas l'accès aux informations d'en-tête, l'appel retourne `null`. Il n'existe pas de méthode `getDateHeaders()` ou `getIntHeaders()`.

L'*exemple 4-19* montre l'utilisation de ces méthodes dans une servlet afin d'afficher des informations sur ses en-têtes de requête HTTP.

Exemple 4-19. Examiner les en-têtes

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HeaderSnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    out.println("En-têtes de la requête :");
    out.println();
    Enumeration names = req.getHeaderNames();
    while (names.hasMoreElements()) {
      String name = (String) names.nextElement();
```

Exemple 4-19. Examiner les en-têtes (suite)

```
      Enumeration values = req.getHeaders(name);  // Support des
                                                  // valeurs multiples.
      if (values != null) {
        while (values.hasMoreElements()) {
          String value = (String) values.nextElement();
          out.println(name + " : " + value);
        }
      }
    }
  }
}
```

Voici un exemple de sortie de cette servlet :

```
En-têtes de la requête :

Connection : Keep-Alive
If-Modified-Since : Thursday, 17-Feb-00 23:23:58 GMT; length=297
User-Agent : Mozilla/4.7 [en] (WinNT; U)
Host : localhost:8080
Accept : image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language : fr
Accept-Language : en
Accept-Charset : iso-8859-1,*,utf-8
Cookie : JSESSIONID=q1886xlc31
```

Examiner le flot d'entrée

Chaque requête traitée par une servlet possède un flot d'entrée associé. De la même façon qu'une servlet peut écrire dans le `PrintWriter` ou le `OutputStream` associés à son objet de réponse, elle peut lire à partir d'un `Reader` ou d'un `InputStream` associés à son objet de requête. Les données lues sur le flot d'entrée peuvent être de n'importe quel type et de n'importe quelle taille. Le flot d'entrée a trois objectifs :

- Passer à une servlet HTTP le contenu associé à une requête POST.
- Passer à une servlet non-HTTP les données brutes envoyées par le client.

Pour lire des données à partir du flot d'entrée, vous devez utiliser la méthode `getReader()` afin d'obtenir un objet `BufferedReader` :

```
public BufferedReader ServletRequest.getReader() throws IOException
```

L'avantage d'utiliser un `BufferedReader` pour lire des données de type caractère est qu'il permet de traduire les jeux de caractères de façon appropriée. Cette méthode lance une exception `IllegalStateException` si `getInputStream()` a été appelée précédemment sur la même requête. Elle lance l'exception `UnsupportedEncodingException` si le codage des caractères de l'entrée n'est pas supporté ou reconnu.

Pour lire des données binaires à partir du flot d'entrée, utilisez `getInputStream()` afin d'obtenir un objet `ServletInputStream` :

```
public ServletInputStream ServletRequest.getInputStream() throws IOException
```

`ServletInputStream` est une sous-classe directe d'`InputStream` et elle peut être traitée comme un `InputStream` normal, avec la possibilité supplémentaire de lire efficacement des données ligne à ligne et de les stocker dans un tableau d'octets. La méthode lance une exception `IllegalStateException` si `getReader()` a été appelée précédemment sur la même requête. Une fois que vous avez le `ServletInputStream`, vous pouvez lire une ligne en utilisant `readLine()` :

```
public int ServletInputStream.readLine(byte b[], int off, int len)
  throws IOException
```

Cette méthode lit des octets à partir du flot d'entrée dans le tableau d'octets `b`, en commençant à l'offset indiqué par `off`. Elle s'arrête de lire lorsqu'elle rencontre un `\n` ou après `len` octets.[5] Le caractère final `\n` est aussi lu dans le tampon. La méthode retourne le nombre d'octets lus ou `-1` si la fin du flot a été atteinte.

Une servlet peut aussi vérifier le type du contenu et la longueur des données envoyées sur le flot d'entrée, en utilisant `getContentType()` et `getContentLength()` :

```
public String ServletRequest.getContentType()
public int ServletRequest.getContentLength()
```

`getContentType()` retourne le type de média du contenu envoyé sur le flot d'entrée ou `null` s'il n'est pas connu (par exemple, s'il n'y a pas de données). `getContentLength()` retourne la longueur, en octets, du contenu envoyé sur le flot d'entrée ou `-1` si elle n'est pas connue.

Traiter les requêtes POST en utilisant le flot d'entrée

Il est rare qu'une servlet traitant une requête POST soit forcée d'utiliser son entrée pour accéder aux données de la requête POST. En général, ces données ne sont rien de plus qu'une information codée dans les paramètres. Une servlet peut donc les retrouver avec la méthode `getParameter()`.

Une servlet peut identifier ce type de requête POST en contrôlant le type du contenu du flot d'entrée. S'il est du type `application/x-www-form-urlencoded`, les données peuvent être retrouvées par `getParameter()` et des méthodes similaires.

Une servlet pourrait souhaiter appeler la méthode `getContentLength()` avant d'appeler `getParameter()` pour éviter une attaque par saturation. Un client mal intentionné pourrait envoyer une quantité énorme de données dans la requête POST, en espérant ralentir le serveur lorsque la méthode `getParameter()` de la servlet s'occupe des données. Par mesure préventive, une servlet peut utiliser `getContentLength()` pour vérifier que la longueur est raisonnable, sans doute moins que 4K.

Recevoir des fichiers en utilisant le flot d'entrée

Une servlet peut aussi recevoir un fichier en utilisant son flot d'entrée. Avant de voir comment, il est important de noter que le chargement de fichier est expérimental et

---

5. Les implémentations de `readLine()` dans l'API Servlet 2.0 souffrent d'un bogue venant du fait que le paramètre `len` est ignoré, conduisant à des problèmes dont le lancement d'une exception `ArrayIndexOutOfBoundsException` si la longueur de la ligne excède la taille du tampon. Ce bogue a été corrigé dans l'API Servlet 2.1 et ultérieure.

qu'il n'est pas supporté par tous les navigateurs. Netscape l'a ajouté dans Netscape Navigator 3 et Microsoft l'a ajouté dans Internet Explorer 4.

La spécification complète du déchargement de fichier est contenu dans la RFC expérimentale 1867, disponible à *http://www.ietf.org/rfc/rfc1867.txt*. Pour résumer rapidement, on peut dire qu'un nombre quelconque de fichiers et de paramètres peuvent être envoyés dans une seule requête POST. La mise en forme de cette requête est différente de celle des données de formulaire `application/x-www-form-urlencoded` classique : son type de contenu est `multipart/form-data`.

Il est assez simple d'écrire la partie client d'un déchargement de fichier. Le code HTML de l'*exemple 4-20* génère un formulaire qui demande d'entrer un nom d'utilisateur et un fichier à décharger. Notez l'ajout de l'attribut `ENCTYPE` et l'utilisation du type d'entrée `FILE` :

Exemple 4-20. Formulaire de sélection d'un fichier à décharger

```
<FORM ACTION="/servlet/UploadTest" ENCTYPE="multipart/form-data" METHOD=POST>
Quel est votre nom ? <INPUT TYPE=TEXT NAME=submitter> <BR>
Quel fichier voulez-vous décharger ? <INPUT TYPE=FILE NAME=file> <BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

Un utilisateur recevant ce formulaire voit une page ressemblant à celle montrée à la *figure 4-3*. Un nom de fichier peut être entré dans la zone de texte ou le fichier peut être choisi en parcourant le système de fichiers. Il est possible d'employer plusieurs champs `<INPUT TYPE=FILE>`, mais les navigateurs actuels n'acceptent de décharger qu'un seul fichier par champ. Après avoir fait la sélection, l'utilisateur envoie le formulaire de façon classique.

Le rôle du serveur pendant le déchargement d'un fichier est plus complexe. Du point de vue de la servlet réceptrice, la soumission n'est rien de plus qu'un flot de données brutes sur son flot d'entrée — un flot de données mis en forme conformément au type de contenu `multipart/form-data` décrit dans la RFC 1867. L'API Servlet ne fournit aucune méthode pour aider à l'analyse des données. Pour vous simplifier la vie (et aussi la nôtre car nous ne voulons pas vous expliquer la RFC 1867), Jason a écrit une classe utilitaire qui fait le travail à votre place. Elle s'appelle `MultipartRequest` et elle est montrée plus loin à l'*exemple 4-22*.

`MultipartRequest` enveloppe `ServletRequest` et présente au programmeur de servlet une API simple. La classe possède deux constructeurs :

```
public MultipartRequest(HttpServletRequest request, String saveDirectory,
                        int maxPostSize) throws IOException
public MultipartRequest(HttpServletRequest request,
                        String saveDirectory) throws IOException
```

Chacune de ces méthodes crée un nouvel objet `MultipartRequest` pour gérer la requête indiquée, sauvegardant tout fichier déchargé dans le répertoire `saveDirectory`. Les deux constructeurs analysent le contenu `multipart/form-data` et lancent une exception `IOException` s'il y a un problème (les servlets utilisant cette classe ne doivent donc pas lire le flot d'entrée elles-mêmes). Le constructeur qui prend un paramètre `maxPostSize` lance aussi une exception `IOException` si le contenu déchargé est plus grand que

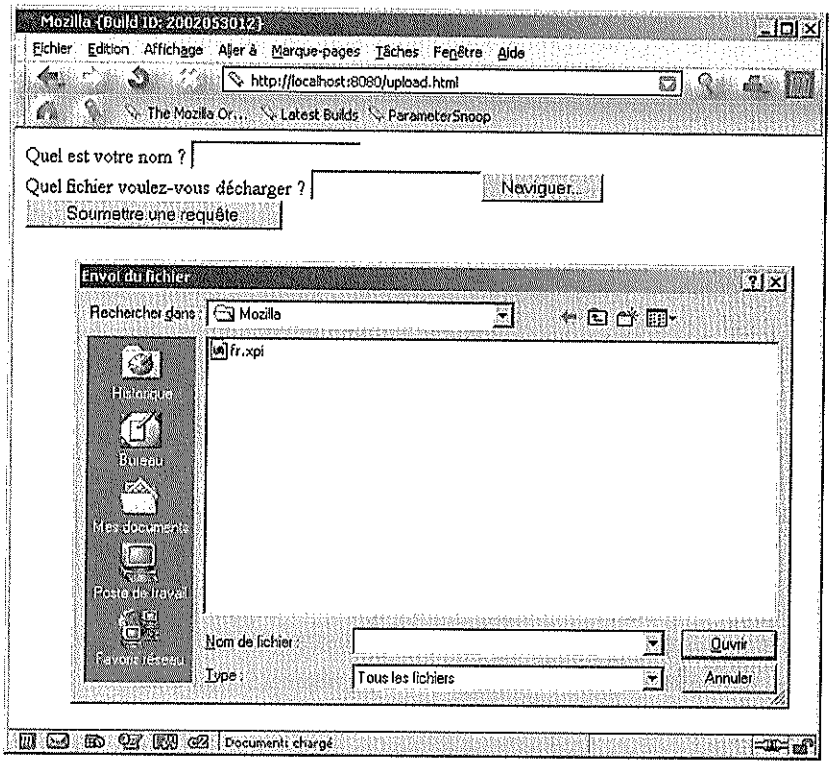


Figure 4-3. Sélection du fichier à décharge

`maxPostSize`. Le second constructeur suppose une valeur par défaut de 1 Mo pour `maxPostSize`.

Notez que le serveur a deux options quand il reçoit un déchargement dont la taille du contenu est trop grande. Premièrement, il peut essayer d'envoyer une page d'erreur, attendre que le client se déconnecte et pendant ce temps lire en silence le contenu déchargé. Cette approche est conforme à la spécification HTTP/1.1 de la RFC 2616, section 8.2.2, qui dit qu'un client doit contrôler l'arrivée d'une page d'erreur pendant le déchargement (voir *http://www.ietf.org/rfc/rfc2616.txt*) et qu'il doit stopper le déchargement quand il reçoit une erreur. Cela garantit que chaque client voit un message d'erreur correct, mais pour les nombreux navigateurs qui n'attendent pas un code d'erreur, cette approche consomme de la bande passante sur le serveur pendant que le client continue son déchargement complet. C'est pourquoi, beaucoup de serveurs implémentent une seconde option : essayer d'envoyer une page d'erreur et forcer la déconnexion si nécessaire. Ainsi, le client reçoit un message poli, mais il stoppe assurément le déchargement.

La classe `MultipartRequest` a sept méthodes publiques qui vous permettent d'obtenir des informations sur la requête. Vous noterez que plusieurs de ces méthodes sont calquées sur celles de `ServletRequest`. Utilisez `getParameterNames()` pour retrouver le nom de tous les paramètres de la requête :

```
public Enumeration MultipartRequest.getParameterNames()
```

Cette méthode retourne le nom de tous les paramètres sous la forme d'un Enumeration d'objets String, ou un Enumeration vide s'il n'y avait pas de paramètres.

Pour obtenir la valeur d'un paramètre nommé, utilisez les méthodes getParameter() ou getParameterValues() :

```
public String MultipartRequest.getParameter(String name)
```

Cette méthode retourne la valeur du paramètre nommé sous forme d'un String ou null si le paramètre n'existait pas. La valeur est garantie être sous sa forme normale décodée. Si le paramètre a de multiples valeurs, seule la dernière est retournée.

```
public String[] MultipartRequest.getParameterValues(String name)
```

Cette méthode retourne toutes les valeurs du paramètre nommé sous forme d'un tableau d'objets String ou null si le paramètre n'existait pas. Une valeur unique est retournée dans un tableau de longueur 1.

Utilisez getFileNames() pour obtenir une liste de tous les fichiers déchargés :

```
public Enumeration MultipartRequest.getFileNames()
```

Cette méthode retourne le nom de tous les fichiers déchargés sous forme d'un Enumeration d'objets String, ou un Enumeration vide s'il n'y a pas de fichier déchargé. Notez que chaque nom de fichier est le nom spécifié dans l'attribut de nom du formulaire HTML, pas par l'utilisateur. Une fois que vous avez le nom d'un fichier, vous pouvez obtenir son nom sur le système de fichiers en utilisant getFilesystemName() :

```
public String MultipartRequest.getFilesystemName(String name)
```

Cette méthode retourne le nom sur le système de fichiers du fichier indiqué ou null si le fichier n'était pas inclus dans le déchargement. Un nom sur le système de fichiers est le nom précisé par l'utilisateur. C'est aussi le nom sous lequel le fichier est réellement sauvé. Vous pouvez accéder au type du contenu du fichier avec getContentType() :

```
public String MultipartRequest.getContentType(String name)
```

Cette méthode retourne le type du contenu du fichier indiqué (comme fourni par le navigateur du client) ou null si le fichier n'était pas inclus dans le déchargement. Enfin, vous pouvez obtenir un objet java.io.File pour le fichier avec getFile() :

```
public File MultipartRequest.getFile(String name)
```

Cette méthode retourne un objet File pour le fichier indiqué sauvé sur le système de fichiers du serveur ou null si le fichier n'était pas inclus dans le déchargement.

L'*exemple 4-21* montre comment une servlet utilise MultipartRequest. La servlet ne fait rien de plus qu'afficher des statistiques sur ce qui est déchargé. Notez qu'elle ne supprime pas les fichiers qu'elle sauve.

Exemple 4-21. Gérer un déchargement de fichier

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.MultipartRequest;
```

Exemple 4-21. Gérer un déchargement de fichier (suite)

```
public class UploadTest extends HttpServlet {

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    try {
      // Croire sur parole que c'est une requête multipart/form-data.

      // Construire un MultipartRequest pour aider à la lecture des informations.
      // Passer la requête, un répertoire pour stocker les fichiers et la
      // taille maximale de POST que nous devons traiter. Ici, nous écrivons
      // sur la racine du serveur et imposons une limite de 5 Mo.
      MultipartRequest multi =
        new MultipartRequest(req, ".", 5 * 1024 * 1024);

      out.println("<HTML>");
      out.println("<HEAD><TITLE>Test de déchargement</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<H1>Test de déchargement</H1>");

      // Afficher les paramètres reçus.
      out.println("<H3>Paramètres :</H3>");
      out.println("<PRE>");
      Enumeration params = multi.getParameterNames();
      while (params.hasMoreElements()) {
        String name = (String)params.nextElement();
        String value = multi.getParameter(name);
        out.println(name + " = " + value);
      }
      out.println("</PRE>");

      // Montrer quels fichiers nous avons reçus.
      out.println("<H3>Fichiers :</H3>");
      out.println("<PRE>");
      Enumeration files = multi.getFileNames();
      while (files.hasMoreElements()) {
        String name = (String)files.nextElement();
        String filename = multi.getFilesystemName(name);
        String type = multi.getContentType(name);
        File f = multi.getFile(name);
        out.println("nom : " + name);
        out.println("nom de fichier : " + filename);
        out.println("type : " + type);
        if (f != null) {
          out.println("longueur : " + f.length());
        }
```

Exemple 4-21. Gérer un déchargement de fichier (suite)

```
        out.println();
      }
      out.println("</PRE>");
    }
    catch (Exception e) {
      out.println("<PRE>");
      e.printStackTrace(out);
      out.println("</PRE>");
    }
    out.println("</BODY></HTML>");
  }
}
```

La servlet passe son objet de requête au constructeur de `MultipartRequest`, ainsi qu'un répertoire relatif à la racine du serveur et dans lequel les fichiers déchargés doivent être stockés (puisque les gros fichiers pourraient ne pas tenir en mémoire) et une taille maximale de 5 Mo. La servlet utilise ensuite `MultipartRequest` pour parcourir les paramètres envoyés. Notez que l'API de gestion des paramètres de `MultipartRequest` correspond à celle de `ServletRequest`. Enfin, la servlet utilise `MultipartRequest` pour parcourir les fichiers envoyés. Pour chaque fichier, elle récupère son nom (spécifié dans le formulaire), le nom sur le système de fichiers (spécifié par l'utilisateur) et le type du contenu. Elle obtient aussi une référence `File` et l'utilise pour afficher la longueur du fichier sauvé. S'il y a un problème, la servlet retourne une exception à l'utilisateur.

L'*exemple 4-22* montre le code de `MultipartRequest`. Vous noterez que la classe `MultipartRequest` utilise en coulisses `com.oreilly.servlet.multipart.MultipartParser` pour l'analyse de la requête. La classe `MultipartParser` fournit un accès de bas niveau au déchargement en parcourant la requête morceau par morceau. Cela permet, par exemple, un déchargement direct de fichiers dans une base de données ou la vérification de certains critères sur un fichier. Le code et la documentation de `MultipartParser` sot disponibles à *http://www.servlets.com*. (Merci à Geoff Soutter pour son travail sur l'analyseur syntaxique.)

Sachez que certains fournisseurs ne testent pas correctement leurs serveurs sur le déchargement de fichiers. Il n'est pas rare que cette classe permette de découvrir des bogues du serveur. Si vous rencontrez un problème lors de l'utilisation de cette classe, essayez un autre serveur (comme Tomcat) afin d'identifier s'il s'agit d'un problème spécifique au serveur et si vous devez alors contacter votre fournisseur.

Exemple 4-22. La classe MultipartRequest

```
package com.oreilly.servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.multipart.MultipartParser;
```

Exemple 4-22. La classe MultipartRequest (suite)

```
import com.oreilly.servlet.multipart.Part;
import com.oreilly.servlet.multipart.FilePart;
import com.oreilly.servlet.multipart.ParamPart;

// Classe utilitaire pour la gestion des requêtes
// <code>multipart/form-data</code>.
public class MultipartRequest {

  private static final int DEFAULT_MAX_POST_SIZE = 1024 * 1024;  // 1 méga.

  private Hashtable parameters = new Hashtable();// nom - vecteur des param.
  private Hashtable files = new Hashtable();     // nom - fichiers déchargés.

  public MultipartRequest(HttpServletRequest request,
                          String saveDirectory) throws IOException {
    this(request, saveDirectory, DEFAULT_MAX_POST_SIZE);
  }

  public MultipartRequest(HttpServletRequest request,
                          String saveDirectory,
                          int maxPostSize) throws IOException {
    // Contrôle des valeurs.
    if (request == null)
      throw new IllegalArgumentException("request ne peut être null");
    if (saveDirectory == null)
      throw new IllegalArgumentException("saveDirectory ne peut être null");
    if (maxPostSize <= 0) {
      throw new IllegalArgumentException("maxPostSize doit être positif");
    }

    // Sauvegarder le répertoire.
    File dir = new File(saveDirectory);

    // Vérifier que saveDirectory est bien un répertoire.
    if (!dir.isDirectory())
      throw new IllegalArgumentException("Pas un répertoire : " +
        saveDirectory);

    // Vérifier que l'on peut écrire dans saveDirectory.
    if (!dir.canWrite())
      throw new IllegalArgumentException("Protégé en écriture : " +
        saveDirectory);

    // Analyser les différentes parties, sauver les fichiers dans le
    // répertoire indiqué et remplir les méta-objets décrivant ce que
    // nous rencontrons.
    MultipartParser parser = new MultipartParser(request, maxPostSize);

    Part part;
```

Exemple 4-22. La classe MultipartRequest (suite)

```
    while ((part = parser.readNextPart()) != null) {
      String name = part.getName();
      if (part.isParam()) {
        // C'est un paramètre, l'ajouter au vecteur de valeurs.
        ParamPart paramPart = (ParamPart) part;
        String value = paramPart.getStringValue();
        Vector existingValues = (Vector)parameters.get(name);
        if (existingValues == null) {
          existingValues = new Vector();
          parameters.put(name, existingValues);
        }
        existingValues.addElement(value);
      }
      else if (part.isFile()) {
        // C'est un fichier.
        FilePart filePart = (FilePart) part;
        String fileName = filePart.getFileName();
        if (fileName != null) {
          // La partie contient un fichier.
          filePart.writeTo(dir);
          files.put(name, new UploadedFile(
                      dir.toString(), fileName, filePart.getContentType()));
        }
        else {
          // Le champ ne contient pas de fichier.
          files.put(name, new UploadedFile(null, null, null));
        }
      }
    }
  }

  // Constructeur avec une ancienne signature, gardé pour des raisons de
  // compatibilité.
  public MultipartRequest(ServletRequest request,
                          String saveDirectory) throws IOException {
    this((HttpServletRequest)request, saveDirectory);
  }

  // Constructeur avec une ancienne signature, gardé pour des raisons de
  // compatibilité.
  public MultipartRequest(ServletRequest request,
                          String saveDirectory,
                          int maxPostSize) throws IOException {
    this((HttpServletRequest)request, saveDirectory, maxPostSize);
  }

  public Enumeration getParameterNames() {
    return parameters.keys();
  }
```

Exemple 4-22. La classe MultipartRequest (suite)

```
  public Enumeration getFileNames() {
    return files.keys();
  }

  public String getParameter(String name) {
    try {
      Vector values = (Vector)parameters.get(name);
      if (values == null || values.size() == 0) {
        return null;
      }
      String value = (String)values.elementAt(values.size() - 1);
      return value;
    }
    catch (Exception e) {
      return null;
    }
  }

  public String[] getParameterValues(String name) {
    try {
      Vector values = (Vector)parameters.get(name);
      if (values == null || values.size() == 0) {
        return null;
      }
      String[] valuesArray = new String[values.size()];
      values.copyInto(valuesArray);
      return valuesArray;
    }
    catch (Exception e) {
      return null;
    }
  }

  public String getFilesystemName(String name) {
    try {
      UploadedFile file = (UploadedFile)files.get(name);
      return file.getFilesystemName();  // Peut être null.
    }
    catch (Exception e) {
      return null;
    }
  }

  public String getContentType(String name) {
    try {
      UploadedFile file = (UploadedFile)files.get(name);
      return file.getContentType();  // Peut être null.
    }
```

Exemple 4-22. La classe MultipartRequest (suite)

```
    catch (Exception e) {
      return null;
    }
  }

  public File getFile(String name) {
    try {
      UploadedFile file = (UploadedFile)files.get(name);
      return file.getFile();  // Peut être null.
    }
    catch (Exception e) {
      return null;
    }
  }
}


// Classe détenant les informations sur un fichier déchargé.
class UploadedFile {

  private String dir;
  private String filename;
  private String type;

  UploadedFile(String dir, String filename, String type) {
    this.dir = dir;
    this.filename = filename;
    this.type = type;
  }

  public String getContentType() {
    return type;
  }

  public String getFilesystemName() {
    return filename;
  }

  public File getFile() {
    if (dir == null || filename == null) {
      return null;
    }
    else {
      return new File(dir + File.separator + filename);
    }
  }
}
```

`MultipartRequest` est utilisable en production et, contrairement à beaucoup d'autres bibliothèques de déchargement, elle supporte les déchargements de taille arbitraire. Savez-vous pourquoi la classe n'implémente pas l'interface `HttpServletRequest` ? Si c'était le cas, on limiterait sa compatibilité ascendante. Si la classe implémentait `HttpServletRequest` et si l'API Servlet 2.3 ajoutait une méthode à l'interface, cette classe n'implémenterait plus complètement l'interface, provoquant l'échec de la compilation des servlets l'utilisant.

Attributs supplémentaires

Parfois, une servlet a besoin de connaître une information sur une requête qui n'est pas disponible *via* les méthodes mentionnées précédemment. Dans ces cas, il existe une dernière solution, la méthode `getAttribute()`. Vous souvenez-vous de la méthode `getAttribute()` de `ServletContext` qui retourne des attributs sur le serveur et qui lui sont spécifiques ? `ServletRequest` a aussi une méthode `getAttribute()` :

```
public Object ServletRequest.getAttribute(String name)
```

Cette méthode retourne la valeur d'un attribut spécifique au serveur pour la requête ou `null` si le serveur ne supporte pas l'attribut de requête précisé. Cette méthode permet à un serveur de fournir à une servlet une information personnalisée sur une requête. Les serveurs sont libres de fournir les attributs qu'ils veulent, et même aucun attribut. Les seules règles sont que le nom des attributs doit suivre la même convention que le nom des paquetages, les noms `java.*` et `javax.*` étant réservés à la division Java Software de Sun Microsystems, et `com.sun.*` étant réservé à Sun Microsystems. Vous devez consulter la documentation de votre serveur pour avoir la liste de ses attributs. N'oubliez pas que l'emploi d'attributs spécifiques à un serveur restreint la portabilité de votre application.

Les servlets peuvent aussi ajouter leurs propres attributs à la requête en utilisant la méthode `setAttribute()` (voir le chapitre 11). La méthode `getAttributeNames()` permet d'obtenir la liste de tous les attributs courants définis par le serveur ou ajoutés par les servlets :

```
public Enumeration ServletRequest.getAttributeNames()
```

Le code suivant affiche tous les attributs courants :

```
Enumeration enum = req.getAttributeNames();
while (enum.hasMoreElements()) {
  String name = (String) enum.nextElement();
  out.println("  req.getAttribute(\"" + name + "\"): " +
              req.getAttribute(name));
}
```

Il existe plusieurs attributs standard qui concernent les requêtes (voir le chapitre 11) et les certificats numériques côté client (voir le chapitre 8).

5

Envoyer une information HTML

Dans le chapitre précédent, nous avons appris qu'une servlet a accès à toutes sortes d'informations — informations sur le client, sur le serveur, sur la requête et même sur elle-même. Maintenant, il est temps de regarder ce qu'une servlet peut faire de ces informations en sachant comment définir et envoyer de l'information.

Ce chapitre commence par revoir comment une servlet retourne une réponse HTML normale, en expliquant complètement certaines méthodes survolées dans les exemples précédents. Ensuite, nous verrons comment diminuer le surcoût impliqué par le retour d'une réponse conservant ouverte la connexion avec le client. Puis, nous examinerons les autres choses que nous pouvons faire avec HTML et HTTP, dont l'envoi des erreurs et autres codes d'état, l'envoi des informations d'en-têtes personnalisés, la redirection de la requête, l'utilisation du *pull* client, la gestion des exceptions d'une servlet, la détection de la déconnexion de l'utilisateur et l'écriture des données dans le journal du serveur.

Contrairement à la première édition, ce chapitre ne détaille pas la génération de HTML dans les servlets. À la place, nous introduisons à la fin de ce chapitre le concept d'architecture de création de contenu qui peut reposer sur les servlets et fournir un moyen plus efficace de générer du contenu HTML. Il s'agit d'une introduction aux chapitres qui présentent individuellement différents cadres.

Structure d'une réponse

Une servlet HTTP peut retourner trois sortes d'informations à un client : un code d'état seul, un nombre quelconque d'en-têtes HTTP et un corps de réponse. Un *code d'état* est une valeur entière qui décrit, comme on peut le supposer, l'état de la réponse. Le code d'état peut indiquer une réussite ou un échec, ou il peut indiquer au logiciel client d'effectuer d'autres actions pour terminer la requête. Le code d'état numérique est souvent accompagné d'une *phrase explicative* qui décrit l'état sous une forme plus compréhensible par un humain. Habituellement, un code d'état travaille en coulisses et il est interprété par le navigateur. Quelquefois, en particulier quand les choses vont mal, le navigateur peut montrer le code d'état à l'utilisateur. Le code le plus connu est proba-

blement le code « 404 Not Found », envoyé par un serveur web qui ne trouve pas l'URL demandée.

Nous avons vu des en-têtes HTTP dans le chapitre précédent lorsque les clients les utilisaient pour envoyer des informations supplémentaires accompagnant la requête. Dans ce chapitre, nous verrons comment une servlet peut envoyer des en-têtes HTTP avec sa réponse.

Le corps de la réponse est le contenu principal de la réponse. Pour une page HTML, le corps de la réponse est le code HTML lui-même. Pour une image, le corps de la réponse contient les octets qui composent l'image. Un corps de réponse peut être de tout type et de n'importe quelle longueur ; le client sait ce qu'il va recevoir en lisant et interprétant les en-têtes HTTP de la réponse.

Une servlet générique est bien plus simple qu'une servlet HTTP — elle retourne simplement un corps de réponse à son client. Cependant, il est possible qu'une sous-classe de `GenericServlet` présente une API qui divise ce code de réponse unique en une structure plus élaborée, donnant l'apparence de retourner de multiples éléments. En fait, c'est exactement ce que fait une servlet HTTP. Au plus bas niveau, un serveur web envoie sa réponse complète au client sous forme d'un flot d'octets. Toute méthode qui définit des codes d'état et des en-têtes n'est qu'une abstraction au-dessus de cela.

Il est important de comprendre cela car même si un programmeur de servlets n'a pas besoin de connaître les détails du protocole HTTP, ce protocole affecte l'ordre dans lequel une servlet peut appeler ses méthodes. Plus précisément, le protocole HTTP impose que les codes d'état et les en-têtes soient envoyés avant le corps de la réponse. Ainsi, une servlet doit faire attention à toujours définir ses codes d'état et ses en-têtes avant de retourner le corps de sa réponse. Une servlet peut mettre dans un tampon le corps de sa réponse afin d'avoir une certaine souplesse, mais une fois le corps de la réponse envoyée, la réponse est considérée comme ayant été *validée* et il n'est plus possible de modifier le code d'état et les en-têtes.

Envoyer une réponse normale

Commençons notre examen des réponses de servlet avec un autre regard sur la première servlet de ce livre, la servlet `HelloWorld`, montrée à l'*exemple 5-1*. Nous espérons qu'elle vous apparaît beaucoup plus simple maintenant qu'au chapitre 2, *Introduction aux servlets HTTP*.

Exemple 5-1. Encore Bonjour

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");
```

Exemple 5-1. Encore Bonjour (suite)

```
    PrintWriter out = res.getWriter();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>Bonjour tout le monde</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<BIG>Bonjour tout le monde</BIG>");
    out.println("</BODY></HTML>");
  }
}
```

Cette servlet utilise deux méthodes et une classe que nous avons rapidement citées précédemment. La méthode `setContentType()` de `ServletResponse` définit le type du contenu de la réponse :

```
public void ServletResponse.setContentType(String type)
```

Dans une servlet HTTP, cette méthode définit l'en-tête HTTP `Content-Type`.

La méthode `getWriter()` renvoie un `PrintWriter` pour l'écriture des données de type caractère de la réponse :

```
public PrintWriter ServletResponse.getWriter() throws IOException
```

L'écrivain code les caractères conformément au jeu de caractères indiqué par le type du contenu. Si aucun jeu de caractères n'est précisé, comme c'est généralement le cas, l'écrivain utilise le codage ISO-8859-1 (Latin-1) approprié aux langues de l'Europe de l'Ouest. Le chapitre 13, *Internationalisation*, décrit en détail les jeux de caractères. Pour le moment, nous nous rappellerons seulement qu'il vaut mieux toujours définir le type de contenu avant d'obtenir un `PrintWriter`. Cette méthode lance une exception `IllegalStateException` si `getOutputStream()` a déjà été appelée pour cette réponse, elle lance une exception `UnsupportedEncodingException` si le codage du flot de sortie n'est pas supporté ou inconnu.

En plus d'utiliser un `PrintWriter` pour retourner une réponse, une servlet peut utiliser une sous-classe spéciale de `java.io.OutputStream` pour écrire des données binaires, la classe `ServletOutputStream` définie dans `javax.servlet`. Vous pouvez obtenir un `ServletOutputStream` avec `getOutputStream()` :

```
public ServletOutputStream ServletResponse.getOutputStream()
    throws IOException
```

Cette méthode retourne un `ServletOutputStream` pour écrire des données de réponse binaires (un octet à la fois). Aucun codage n'est effectué. Cette méthode lance une exception `IllegalStateException` si `getWriter()` a déjà été appelée pour cette réponse.

La classe `ServletOutputStream` ressemble à la classe Java standard `PrintStream`. Dans l'API Servlet Version 1.0, cette classe était utilisée pour toute sortie d'une servlet, qu'elle soit textuelle ou binaire. Cependant, dans l'API Servlet API Version 2.0 et ultérieure, son rôle se limite aux données de sortie binaires. En tant que sous-classe directe de `OutputStream`, elle met à disposition les méthodes `write()`, `flush()` et `close()` de la classe `OutputStream`. Elle y ajoute ses propres méthodes `print()` et `println()` pour écrire la plupart des types de données primitifs de Java (voir l'annexe A, *L'API Servlet, référence rapide*, pour une liste complète). La seule différence entre l'interface de `ServletOutput-`

Stream et celle de PrintStream est que les méthodes print() et println() de ServletOutputStream ne peuvent pas, de façon automatique, imprimer directement les paramètres de type Object ou char[].

Utiliser des connexions persistantes

Les *connexions persistantes* (parfois appelées *connexions « keep-alive »*) sont utilisées pour optimiser la façon dont la servlet retourne le contenu au client. Pour comprendre comment fonctionne cette optimisation, vous devez d'abord comprendre comment les connexions HTTP fonctionnent. Nous resterons à un niveau élevé et ne descendrons que lorsqu'il est nécessaire d'expliquer les idées de base. Vous trouverez tous les détails dans le livre de Clinton Wong, *HTTP — précis et concis* (Éditions O'Reilly).

Quand un client, comme un navigateur, demande un document web à partir d'un serveur, il commence par établir une connexion par socket avec ce serveur. Sur cette connexion, le client effectue sa requête et reçoit ensuite la réponse du serveur. Le client indique qu'il a terminé sa requête en envoyant une ligne vide ; le serveur, à son tour, indique que la réponse est complète en fermant la connexion.

Jusqu'ici ça va. Mais que se passe-t-il si la page contient des balises <IMG> ou <APPLET> qui font que le client doit retrouver un autre contenu à partir du serveur ? Et bien, une autre connexion par socket est utilisée. Si une page contient 10 images et une applet composée de 25 classes, 36 connexions sont nécessaires pour le transfert de la page. Maintenant vous savez pourquoi certaines personnes disent que WWW signifie World Wide Wait ! Cette approche équivaut à commander une pizza en faisant des commandes séparées pour chaque ingrédient.

Une meilleure approche consiste à utiliser la même connexion pour retrouver plusieurs parties de la page, ce que l'on appelle une connexion persistante. L'astuce dans une connexion persistante est que le client et le serveur doivent se mettre d'accord sur l'endroit où la réponse du serveur se termine et celui où la prochaine requête du client commence. Ils pourraient essayer d'utiliser une indication basée sur la ligne vide, mais que se passe-t-il alors si la réponse contient elle-même une ligne vide ? Dans une connexion persistante, le serveur indique simplement au client la taille de la réponse dans l'en-tête Content-Length. Le client sait ensuite quand il peut reprendre le contrôle de la socket.

La plupart des serveurs gèrent de façon interne l'en-tête Content-Length pour les fichiers statiques qu'ils envoient ; la valeur de Content-Length correspond à la taille du fichier. Pour déterminer la longueur d'un contenu généré par une servlet, un serveur a besoin de l'aide de la servlet. Une servlet peut fixer la valeur de Content-Length et tirer parti d'une connexion persistante pour son contenu dynamique en utilisant la méthode setContentLength() :

```
public void ServletResponse.setContentLength(int len)
```

Cette méthode définit la longueur (en octets) du contenu qui sera retourné par le serveur. Dans une servlet HTTP, la méthode définit l'en-tête HTTP Content-Length. Notez que l'utilisation de cette méthode est optionnelle. Cependant, si vous l'utilisez, vos servlets seront capables de tirer parti des connexions persistantes lorsqu'elles sont disponibles. Le client sera aussi capable d'afficher une barre de progression pendant le téléchargement.

Si vous invoquez la méthode `setContentLength()`, voici deux avertissements : une servlet doit appeler cette méthode avant d'envoyer le corps de la réponse et la taille donnée doit être exacte. Le moindre petit décalage, même d'un octet, entraînera probablement des problèmes.

Mettre la réponse dans un tampon

À partir de l'API Servlet 2.2, une servlet peut contrôler si le serveur met ou non sa réponse dans un tampon et elle peut définir la taille de ce tampon. Dans les versions précédentes de l'API, la plupart des serveurs web implémentaient la mise en tampon de la réponse afin d'améliorer les performances ; la taille de ce tampon dépendait du serveur. En général, les serveurs disposaient de tampon de 8K.

Un tampon de réponse permet à une servlet d'écrire une partie de sa sortie tout en étant certaine qu'elle ne sera pas envoyée immédiatement. Si la servlet rencontre une erreur, le code d'état et les en-têtes peuvent encore être modifiés, et cela tant que le tampon n'a pas été vidé.

La mise en tampon de la réponse apporte aussi un moyen simple d'éviter les difficultés de calcul de la taille du contenu. Une servlet utilise la mise en tampon pour calculer automatiquement la longueur du contenu, comme le montre l'*exemple 5-2*.

Exemple 5-2. Servlet utilisant la mise en tampon pour supporter les connexions persistantes

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class KeepAlive extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");

    // Demander un tampon de 16K octets ; ne pas fixer la longueur du
    // contenu.
    res.setBufferSize(16 * 1024);

    PrintWriter out = res.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Bonjour tout le monde</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<BIG>Une réponse de taille inférieure à 16K</BIG>");
    out.println("</BODY></HTML>");
  }
}
```

Cette servlet appelle `setBufferSize()` pour demander un tampon de réponse de taille minimale de 16K octets. Puis elle envoie son corps de réponse comme d'habitude. Le serveur utilise un tampon interne d'au moins 16 384 octets pour contenir le corps de la

réponse et attend que le tampon soit plein ou que la servlet lui demande de vider le tampon avant d'envoyer le contenu au client. Si la réponse tient entièrement dans le tampon, le serveur peut (mais n'est pas obligé) définir automatiquement l'en-tête de réponse `Content-Length`.

Il est important de noter que la mise en tampon des réponses a un prix. Mettre en tampon toute la sortie et l'envoyer en une seule fois consomme de la mémoire et peut retarder le moment où le client commence à recevoir les données. Pour les servlets ayant des réponses courtes, les connexions persistantes ont un sens. Mais pour les servlets dont la réponse est longue, le surcoût mémoire et le délai sont probablement plus pénalisants que l'ouverture de quelques connexions supplémentaires.

Il est aussi important de noter que les connexions persistantes ne sont pas supportées par tous les serveurs et tous les clients. Cela dit, il est toujours valable qu'une servlet fixe la longueur de son contenu ou mette sa sortie dans un tampon pour que le serveur puisse en déterminer la taille. L'information de longueur du contenu est utilisée par les serveurs et les clients qui supportent les connexions persistantes, elle est ignorée par les autres.

Contrôler le tampon de réponse

Le contrôle du tampon de réponse se fait par le biais de cinq méthodes de `ServletResponse`. `setBufferSize()` indique au serveur la taille minimale du tampon (en octets) acceptée par la servlet :

```
public void ServletResponse.setBufferSize(int size)
  throws IllegalStateException
```

Le serveur peut réserver un tampon plus grand que celui demandé ; il peut vouloir garder les tampons dans des blocs de 8K, par exemple pour faciliter leur réutilisation. Un tampon plus important permet d'écrire plus de contenu avant que quelque chose soit envoyé. Ainsi, la servlet dispose de plus de temps pour définir ses codes d'état et les entêtes. Un tampon plus petit diminue l'encombrement mémoire du serveur et permet aux clients de recevoir plus tôt les données. Cette méthode doit être appelée avant le début de l'écriture du corps de la réponse ; si le contenu a déjà été écrit, elle lance une exception `IllegalStateException`. Utilisez `getBufferSize()` pour déterminer la taille du tampon :

```
public int ServletResponse.getBufferSize()
```

Cette méthode renvoie un `int` indiquant la taille courante du tampon ou 0 dans le cas très improbable où aucun tampon n'est utilisé.

Vous pouvez appeler `isCommitted()` pour déterminer si une partie de la réponse a été envoyée :

```
public boolean ServletResponse.isCommitted()
```

Si cette méthode retourne `true`, il est trop tard pour modifier le code d'état et les entêtes, et la valeur de `Content-Length` ne peut plus être calculée automatiquement.

Si vous devez reconstruire votre réponse, appelez `reset()` afin d'effacer le tampon de réponse et annuler le code d'état et les en-têtes définis :

```
public void ServletResponse.reset() throws IllegalStateException
```

Cette méthode doit être appelée avant que la réponse soit validée, sinon elle lance une exception `IllegalStateException`. Les méthodes `sendError()` et `sendRedirect()`, présentées plus loin, ont un comportement similaire et elles effacent le tampon de réponse. Cependant, elles ne touchent pas aux en-têtes de réponse.

Vous pouvez aussi forcer l'écriture sur le client du contenu du tampon en appelant `flushBuffer()`. Le client commence alors à recevoir immédiatement le contenu :

```
public void ServletResponse.flushBuffer() throws IOException
```

L'appel de cette méthode valide automatiquement la réponse. L'écriture du code d'état et des en-têtes sont alors écrits et un appel à `reset()` n'est plus possible.

L'*exemple 5-3* montre une servlet qui utilise la méthode `reset()` afin d'écrire puis d'effacer le contenu. Elle affiche la taille par défaut du tampon dans le journal au lieu de l'envoyer au client afin qu'il ne puisse plus être annulé.

Exemple 5-3. Gestion du tampon de réponse

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Buffering extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setBufferSize(8 * 1024); // Tampon de 8K.
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    int size = res.getBufferSize(); // Renvoie 8096 ou plus.

    // Enregistrer la taille par défaut, dans le journal.
    log("La taille par défaut du tampon est " + size);

    out.println("Le client ne verra pas ceci");
    res.reset();
    out.println("Ni cela !");
    res.reset();
    out.println("Et ceci ne sera pas vu si sendError() est appelée");
    if (req.getParameter("important_parameter") == null) {
      res.sendError(res.SC_BAD_REQUEST, "important_parameter obligatoire");
    }
  }
}
```

Codes d'état

Jusqu'à présent, nos exemples de servlets n'ont pas défini de codes d'état HTTP pour la réponse. Nous avons tiré parti du fait que si une servlet ne définit pas de façon spécifique le code d'état, le serveur prend le relais et met un code d'état avec pour valeur par

défaut 200 OK. C'est une facilité pratique lorsque nous retournons des réponses réussies normales. Cependant, en utilisant les codes d'état, une servlet peut en obtenir plus de sa réponse. Par exemple, elle peut rediriger une requête ou rapporter un problème.

Les numéros de codes d'état les plus courants sont définis par des constantes mnémoniques (des champs `public final static int`) dans la classe `HttpServletResponse`. Le *tableau 5-1* en donne quelques-uns. La liste complète se trouve à l'annexe D, *Codes d'état HTTP*.

Tableau 5-1. Codes d'état HTTP

Constante mnémonique	Code	Message par défaut	Signification
`SC_OK`	200	OK	La requête du client a réussi et la réponse du serveur contient les données demandées. C'est le code d'état par défaut.
`SC_NO_CONTENT`	204	No Content	La requête a réussi, mais il n'y avait pas de nouveau corps de réponse à retourner. Les navigateurs recevant ce code doivent garder la représentation courante du document. Ce code est utile pour les servlets lorsqu'elles acceptent des données d'un formulaire mais souhaitent que le navigateur reste sur le formulaire, car il évite le message d'erreur « Document contains no data ».
`SC_MOVED_PERMANENTLY`	301	Moved Permanently	La ressource demandée a été déplacée de façon permanente. Les futures références devraient utiliser la nouvelle URL dans les requêtes. Le nouvel emplacement est donné par l'en-tête `Location`. La plupart des navigateurs accèdent automatiquement au nouvel emplacement.
`SC_MOVED_TEMPORARILY`	302	Moved Temporarily	La ressource demandée a été déplacée temporairement, mais les futures références doivent continuer à utiliser l'URL originale pour y accéder. Le nouvel emplacement est donné par l'en-tête `Location`. La plupart des navigateurs accèdent automatiquement au nouvel emplacement.

Tableau 5-1. Codes d'état HTTP (suite)

Constante mnémonique	Code	Message par défaut	Signification
SC_UNAUTHORIZED	401	Unauthorized	La requête n'a pas les bonnes autorisations. Utilisé avec les en-têtes WWW-Authenticate et Authorization.
SC_NOT_FOUND	404	Not Found	La ressource demandée n'a pas été trouvée ou n'est pas disponible.
SC_INTERNAL_SERVER_ERROR	500	Internal Server Error	Une erreur inattendue est survenue dans le serveur et elle l'empêche de satisfaire la requête.
SC_NOT_IMPLEMENTED	501	Not Implemented	Le serveur ne supporte pas la fonctionnalité nécessaire pour satisfaire la requête.
SC_SERVICE_UNAVAILABLE	503	Service Unavailable	Le service (serveur) est temporairement indisponible mais devrait être prochainement rétabli. Si le serveur sait quand il sera à nouveau disponible, l'en-tête Retry-After peut être fourni.

Mettre un code d'état

Une servlet utilise `setStatus()` pour définir le code d'état d'une réponse :

```
public void HttpServletResponse.setStatus(int sc)
```

Cette méthodes établit le code d'état HTTP à la valeur donnée. Le code peut être indiqué par un numéro ou par l'un des codes `SC_XXX` définis dans `HttpServletResponse`. Souvenez-vous que la méthode `setStatus()` doit être appelée avant que la réponse ne soit validée ou bien l'appel est ignoré.

Si une servlet définit un code d'état indiquant une erreur lors du traitement de la requête, elle peut appeler `sendError()` au lieu de `setStatus()` :

```
public void HttpServletResponse.sendError(int sc)
  throws IOException, IllegalStateException
public void HttpServletResponse.sendError(int sc, String sm)
  throws IOException, IllegalStateException
```

La méthode `sendError()` conduit le serveur à générer et à envoyer une page qui lui est propre et qui décrit l'erreur. La page d'erreur de la servlet a ainsi une apparence similaire aux autres pages d'erreur du serveur. L'appel à `setStatus()` suite à une erreur laisse à la servlet la responsabilité de générer la page d'erreur. Lorsque la version à deux arguments est utilisée, le paramètre de message d'état peut être mis directement dans le corps de la réponse, en fonction de l'implémentation du serveur. Cette méthode doit être appelée avant que la réponse ne soit validée ou elle lance une exception `Illegal-`

StateException. Cette méthode effectue une remise à zéro implicite du tampon de réponse avant de générer la page d'erreur. Les en-têtes définis avant sendError() ne changent pas.

Améliorer ViewFile avec les codes d'état

Jusque-là, nous ne nous sommes pas embêtés à appeler ces méthodes pour définir le code d'état d'une réponse. Nous nous sommes simplement reposés sur le fait que le code d'état est par défaut SC_OK. Mais, une servlet a parfois besoin de retourner une réponse dont le code d'état n'est pas SC_OK — lorsque la réponse ne contient pas les données demandées. Prenons pour exemple le traitement d'une exception FileNotFoundException par la servlet ViewFile du chapitre 4, *Retrouver l'information* :

```
// Retourner le fichier.
try {
  ServletUtils.returnFile(file, out);
}
catch (FileNotFoundException e) {
  out.println("Fichier non trouvé");
}
```

Sans définir un code d'état, le mieux que cette servlet puisse faire est d'écrire une explication du problème et d'envoyer cette explication dans la page censée contenir les données du fichier. En revanche, avec un code d'état, elle peut faire exactement ce que la servlet DefaultServlet de Sun fait : mettre le code de la réponse à SC_NOT_FOUND afin d'indiquer que le fichier demandé n'a pas été trouvé et qu'il ne peut être retourné. Voici la version améliorée :

```
// Retourner le fichier.
try {
  ServletUtils.returnFile(file, out);
}
catch (FileNotFoundException e) {
  res.sendError(res.SC_NOT_FOUND);
}
```

L'apparence de la page générée par l'appel à sendError() dépend du serveur, mais elle ressemble en général à la page d'erreur normale du serveur. Pour le serveur Apache/Tomcat, cet appel génère la page 404 Not Found qui lui est propre, complétée par le base de page d'Apache (voir *figure 5-1*). Notez que cette page n'est pas différenciable des autres pages 404 Not Found d'Apache, donnant à la servlet la possibilité de se fondre dans le serveur.

En-têtes HTTP

Une servlet peut utiliser les en-têtes HTTP pour fournir des informations supplémentaires sur sa réponse. Comme nous l'avons dit au chapitre 4, une discussion complète de tous les en-têtes HTTP 1.0 et HTTP 1.1 dépasse le cadre de ce livre. Le *tableau 5-2* donne la liste des en-têtes HTTP le plus souvent utilisés dans la réponse des servlets.

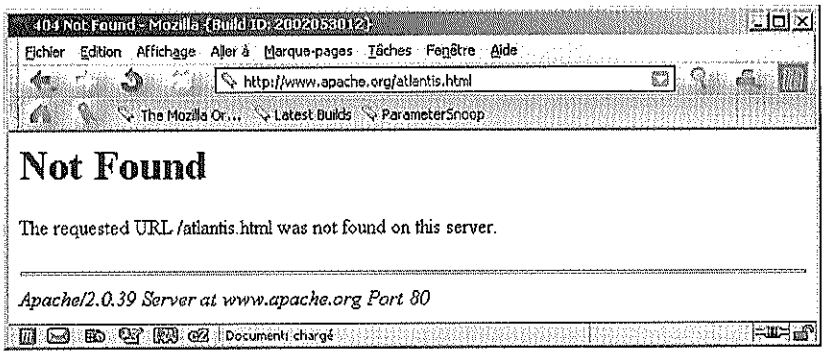


Figure 5-1. La page « 404 Not Found » d'Apache/Tomcat

Tableau 5-2. En-têtes de réponse HTTP

En-tête	Utilisation
`Cache-Control`	Indique tout traitement spécifique qu'un système de cache doit apporter à ce document. Les valeurs les plus courantes sont `no-cache` (pour indiquer que ce document ne doit pas être mis dans le cache), `no-store` (pour indiquer que ce document ne doit pas être mis dans le cache ni même stocké par un serveur proxy, habituellement du fait de son contenu sensible), et `max-age=`*`secondes`* (pour indiquer combien de temps ce document reste valable). Cet en-tête a été introduit dans HTTP 1.1.[a]
`Pragma`	L'équivalent HTTP 1.0 de `Cache-control`, avec `no-cache` comme valeur possible. Il est souhaitable d'utiliser `Pragma` conjointement à `Cache-Control` afin de supporter les navigateurs anciens.
`Connection`	Utilisé pour indiquer si le serveur est disposé à maintenir une connexion ouverte (persistante) avec le client. Si c'est le cas, la valeur est `keep-alive`. Si ce n'est pas le cas, la valeur est `close`. La plupart des serveurs web traitent cet en-tête pour le compte de leurs servlets, en mettant automatiquement la valeur `keep-alive` lorsqu'une servlet définit son en-tête `Content-Length` ou quand le serveur peut automatiquement déterminer la longueur du contenu.
`Retry-After`	Indique un temps après lequel le serveur peut retraiter les requêtes, utilisé avec le code d'état `SC_SERVICE_UNAVAILABLE`. Sa valeur est soit un `int` qui représente le nombre de secondes soit une date qui représente un temps réel.
`Expires`	Indique le moment où le document peut changer ou lorsque son information deviendra non valide. Cela implique qu'il y a peu de chances que le document change avant ce moment.
`Location`	Indique un nouvel emplacement du document, habituellement utilisé avec les codes d'état `SC_CREATED`, `SC_MOVED_PERMANENTLY` et `SC_MOVED_TEMPORARILY`. Sa valeur doit être une URL complète (incluant `http://`).

Tableau 5-2. En-têtes de réponse HTTP (suite)

En-tête	Utilisation
`WWW-Authenticate`	Indique le schéma et le domaine d'autorisation requis pour que le client puisse accéder à l'URL demandée. Utilisé avec le code d'état `SC_UNAUTHORIZED`.
`Content-Encoding`	Indique le schéma utilisé pour coder le corps de la réponse. Les valeurs possibles sont `gzip` (ou `x-gzip`) et `compress` (ou `x-compress`). Les codages multiples sont représentés par une liste de codages séparés par des virgules et donnés dans l'ordre dans lequel ils ont été appliqués aux données.

a. Netscape Navigator et Microsoft Internet Explorer ont des bogues qui empêchent de prendre toujours en compte cet en-tête. Si vous devez absolument garantir que le contenu de la servlet n'est pas mis dans un cache, chaque requête à la servlet doit utiliser une URL un peu différente, peut-être par l'ajout d'une petite une information dans les paramètres.

Donner une valeur à un en-tête HTTP

La classe `HttpServletResponse` fournit des méthodes pour aider les servlets à définir les en-têtes HTTP d'une réponse. La méthode `setHeader()` définit la valeur d'un en-tête :

```
public void HttpServletResponse.setHeader(String name, String value)
```

Cette méthode définit la valeur de l'en-tête précisé sous forme d'un `String`. Le nom est sensible à la casse, comme c'est le cas pour toutes ces méthodes. Si l'en-tête a déjà été défini, la nouvelle valeur remplace la précédente. Tout type d'en-tête peut être défini avec cette méthode.

Si vous devez spécifier une date pour un en-tête, utilisez `setDateHeader()` :

```
public void HttpServletResponse.setDateHeader(String name, long date)
```

Cette méthode met la valeur de l'en-tête indiqué à une date et une heure particulières. La valeur de la date est un `long` qui représente le nombre de millisecondes écoulées depuis l'origine du temps (1er janvier 1970 GMT à minuit). Si l'en-tête a déjà été défini, la nouvelle valeur remplace la précédente.

Enfin, vous pouvez utiliser `setIntHeader()` pour donner une valeur entière à un en-tête :

```
public void HttpServletResponse.setIntHeader(String name, int value)
```

Cette méthode met la valeur de l'en-tête indiqué à partir d'un `int`. Si l'en-tête a déjà été défini, la nouvelle valeur remplace la précédente.

La méthode `containsHeader()` offre une façon de contrôler si l'en-tête existe déjà :

```
public boolean HttpServletResponse.containsHeader(String name)
```

Cette méthode retourne `true` si l'en-tête indiqué a déjà été défini, sinon `false`.

Dans les rares cas où une servlet envoie de multiples valeurs pour le même en-tête, il existe les méthodes suivantes :

```
public void HttpServletResponse.addHeader(String name, String value)
public void HttpServletResponse.addDateHeader(String name, long value)
public void HttpServletResponse.addIntHeader(String name, int value)
```

Ces méthodes donnent à l'en-tête la valeur indiquée, mais tandis que la méthode setHeader() remplace la ou les valeurs existantes, addHeader() conserve les valeurs existantes et ajoute la nouvelle.

Enfin, la spécification HTML 3.2 définit une autre façon de spécifier la valeur des en-têtes en utilisant la balise <META HTTP-EQUIV> dans la page HTML elle-même :

```
<META HTTP-EQUIV="name" CONTENT="value">
```

Cette balise doit faire partie de la section <HEAD> de la page HTML. Cette technique n'apporte aucun bénéfice spécial aux servlets. Elle a été développée pour une utilisation avec des documents statiques n'ayant pas accès à leurs propres en-têtes.

Rediriger une requête

Une des choses utiles qu'une servlet peut faire en utilisant les codes d'état et les en-têtes est de rediriger une requête. Cela se fait en envoyant des instructions au client afin qu'il utilise une autre URL. La redirection est généralement utilisée lorsqu'un document a été déplacé (pour donner au client le nouvel emplacement), pour l'équilibrage de charge (une URL peut ainsi distribuer la charge sur plusieurs machines différentes) ou pour une simple fonction aléatoire (choisir une destination au hasard).

L'*exemple 5-4* montre une servlet qui effectue une redirection aléatoire, envoyant le client vers un site sélectionné aléatoirement dans une liste. En fonction de la liste des sites, une servlet comme celle-ci peut avoir plusieurs utilisations. Telle qu'elle est actuellement, ce n'est qu'un point de passage vers une sélection de sites sympas. Avec une liste de sites contenant des images publicitaires, elle peut être utilisée pour choisir la bannière suivante.

Exemple 5-4. Redirecteur aléatoire

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SiteSelector extends HttpServlet {

  Vector sites = new Vector();
  Random random = new Random();

  public void init() throws ServletException {
    sites.addElement("http://www.oreilly.com/catalog/jservlet2");
    sites.addElement("http://www.servlets.com");
    sites.addElement("http://java.sun.com/products/servlet");
    sites.addElement("http://www.newInstance.com");
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
```

Exemple 5-4. Redirecteur aléatoire (suite)

```
    int siteIndex = Math.abs(random.nextInt()) % sites.size();
    String site = (String)sites.elementAt(siteIndex);

    res.setStatus(res.SC_MOVED_TEMPORARILY);
    res.setHeader("Location", site);
  }
}
```

La redirection se fait en deux lignes :

```
res.setStatus(res.SC_MOVED_TEMPORARILY);
res.setHeader("Location", site);
```

La première ligne définit le code d'état pour indiquer qu'une redirection va avoir lieu, la seconde ligne indique le nouvel emplacement. Pour garantir que cela fonctionne, vous devez appeler ces méthodes avant que la réponse ne soit validée. Souvenez-vous que le protocole HTTP envoie les codes d'état et les en-têtes avant le corps du contenu. Le nouveau site doit aussi être donné sous forme d'une URL absolue (par exemple, *http://serveur:port/path/file.html*). Le non-respect de ces règles peut perturber le client.

Il est possible de remplacer ces deux lignes par une seule en utilisant la méthode utilitaire `sendRedirect()` :

```
public void HttpServletResponse.sendRedirect(String location)
  throws IOException, IllegalStateException
```

Pour notre exemple, les deux lignes deviennent simplement :

```
res.sendRedirect(site);
```

Cette méthode redirige la réponse vers l'emplacement spécifié, définissant automatiquement le code d'état et l'en-tête `Location`. De plus, pour supporter les clients sans possibilité de redirection ou ne reconnaissant pas le code d'état `SC_MOVED_TEMPORARILY`, la méthode écrit un petit corps de réponse qui contient un lien vers le nouvel emplacement. Par conséquent, vous ne devez pas écrire votre propre corps de réponse quand vous utilisez cette méthode. La méthode `sendRedirect()` peut, à partir de l'API Servlet 2.2, accepter une URL relative. La spécification HTTP stipule que toutes les URL de redirection soient des URL absolues ; cette méthode transforme une URL relative en une URL absolue (en ajoutant automatiquement le protocole, le serveur et le port courants, mais pas le chemin de contexte que vous devez ajouter vous-même si nécessaire) avant de l'envoyer au client. Plutôt malin, non ? Assurez-vous d'appeler cette méthode avant que la réponse ne soit validée, sinon elle lancera une exception `IllegalStateException`. Cette méthode effectue une remise à zéro implicite du tampon de réponse avant de générer la page de redirection. Les en-têtes définis avant `sendError()` ne changent pas.

Surveiller les liens vers d'autres sites

La redirection peut aussi servir à savoir où des clients vont quand ils quittent votre site. Supposez que vous ayez plusieurs pages contenant des liens vers d'autres sites. Au lieu de pointer directement vers un site extérieur, vous pouvez aller vers une servlet de redirection qui enregistre la sélection des sites extérieurs. Voici le code HTML :

```
<a href="/goto/http://www.servlets.com">Servlets.com</a>
```

Il est possible d'enregistrer une servlet afin qu'elle gère le préfixe de chemin */goto/** dans lequel elle reçoit l'URL sélectionnée sous forme d'information de chemin supplémentaire. Elle redirige ensuite le client vers cet emplacement après avoir ajouté une trace dans le journal du serveur. Le code de la servlet `GoTo` est donné à l'*exemple 5-5*.

Exemple 5-5. Où pensez-vous aller ?

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GoTo extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // Déterminer le site vers lequel aller.
    String site = req.getPathInfo();
    String query = req.getQueryString();

    // Gérer une requête erronée.
    if (site == null) {
      res.sendError(res.SC_BAD_REQUEST,
                    "Information de chemin supplémentaire nécessaire");
    }

    // Enlever le "/" de tête et ajouter la chaîne d'interrogation. Nous
    // supposons que l'URL de l'information de chemin est absolue.
    String url = site.substring(1) + (query == null ? "" : "?" + query);

    // Journaliser l'URL demandée et rediriger.
    log(url);  // Ou écrire dans un fichier spécial.
    res.sendRedirect(url);
  }
}
```

Une servlet équivalente à celle-ci peut gérer les alias afin que */goto/servlets* aille directement à *http://www.servlets.com*. La vente de tels alias est le modèle commercial de *http://go.to* et *http://i.am*.

Pull client

Le *pull client* est similaire à la redirection, avec une différence principale : le navigateur affiche le contenu de la première page et attend un certain temps avant de retrouver et afficher le contenu de la page suivante. On appelle cela le *pull client* car c'est au client de tirer à lui le contenu de la page suivante.

En quoi cela est-il utile ? Pour deux raisons. D'abord, le contenu de la première page peut expliquer au client que la page demandée a été déplacée avant que la page suivante

ne soit automatiquement chargée. Ensuite, les pages peuvent être retrouvées en séquence, rendant ainsi possible un effet de mouvement.

L'information de pull client est envoyée au client *via* l'en-tête HTTP `Refresh`. La valeur de cet en-tête indique le nombre de secondes pendant lesquelles la page doit être affichée avant d'aller chercher la prochaine, et elle peut aussi inclure l'URL indiquant où aller la chercher. Si aucune URL n'est donnée, la même URL est utilisée. Voici un appel à `setHeader()` qui indique au client de recharger la même servlet après avoir affiché son contenu courant pendant trois secondes :

```
setHeader("Refresh", "3");
```

Et voici un appel qui indique au client d'afficher la page d'accueil de Netscape après trois secondes :

```
setHeader("Refresh", "3; URL=http://home.netscape.com");
```

L'*exemple 5-6* montre une servlet qui utilise le pull client pour afficher l'heure, mise à jour toutes les 10 secondes.

Exemple 5-6. L'heure courante, mise à jour

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ClientPull extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    res.setHeader("Refresh", "10");
    out.println(new Date().toString());
  }
}
```

C'est un exemple d'animation orientée texte (nous verrons des animations graphiques dans le prochain chapitre). Notez que l'en-tête `Refresh` n'est pas répétitif. Ce n'est pas une directive permettant de charger de façon répétitive le document. Cependant, dans cet exemple, l'en-tête `Refresh` est spécifié à chaque rechargement, créant ainsi un affichage continu.

L'utilisation du pull client pour retrouver un second document est montré à l'*exemple 5-7*. Cette servlet redirige les requêtes d'une machine vers une autre machine, en donnant auparavant une explication au client.

Exemple 5-7. Un changement de machine expliqué

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
```

Exemple 5-7. Un changement de machine expliqué (suite)

```
import javax.servlet.http.*;

public class ClientPullMove extends HttpServlet {

  static final String NEW_HOST = "http://www.oreilly.com";

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    String newLocation = NEW_HOST + req.getRequestURI();

    res.setHeader("Refresh", "10; URL=" + newLocation);

    out.println("L'URI demandé a été déplacé sur une autre machine.<BR>");
    out.println("Son nouvel emplacement est " + newLocation + "<BR>");
    out.println("Votre navigateur vous y emmènera dans 10 secondes.");
  }
}
```

Cette servlet génère le nouvel emplacement à partir de l'URI demandé, ce qui lui permet de rediriger toute requête effectuée sur l'ancien serveur vers le même chemin sur le nouveau serveur. Avec le descripteur de déploiement *web.xml*, cette servlet pourrait être configurée pour traiter chaque requête, pour amener progressivement les clients vers le nouvel emplacement.

Lorsque tout va mal

OK, faisons face ! Parfois les choses vont mal. Il arrive qu'un chien morde et qu'une abeille pique. Il y a de nombreuses causes possibles : de mauvais paramètres, des ressources manquantes et (oups !) de vrais bogues. Le point ici est qu'une servlet doit être préparée aux problèmes, tant ceux qui sont attendus que les autres. Deux aspects sont concernés lorsque tout va mal :

- limiter les dégâts sur le serveur,
- informer correctement le client.

Puisque les servlets sont écrites en Java, les dommages potentiels qu'elles peuvent causer à leur serveur sont vraiment minimes. Un serveur peut embarquer (même dans son processus) de façon sûre des servlets, exactement comme un navigateur Web peut embarquer des applets téléchargées. Cette sûreté repose sur les fonctionnalités de sécurité de Java, dont l'utilisation protégée de la mémoire, la gestion des exceptions et les gestionnaires de sécurité. La protection mémoire de Java garantit que les servlets ne pourront accéder accidentellement (ou intentionnellement) aux données internes du serveur. La gestion des exceptions de Java permet au serveur d'intercepter toute exception lancée par une servlet. Même si une servlet effectue accidentellement une division par zéro ou appelle une méthode sur un objet nul, le serveur continue de fonctionner.

Le mécanisme Java de gestion de la sécurité fournit au serveur un moyen de placer les servlets douteuses dans un « aquarium », limitant ainsi leurs fonctions et les empêchant de provoquer intentionnellement des problèmes.

Vous devez être conscient que les servlets de confiance s'exécutant hors de l'aquarium du gestionnaire de sécurité ont la possibilité de causer des dommages au serveur. Par exemple, une servlet peut effacer la zone de fichiers du serveur ou même appeler `System.exit()`. Il est aussi vrai qu'une servlet de confiance ne doit jamais causer des dommages, sauf par accident, et il est difficile d'appeler accidentellement `System.exit()`. De plus, même les servlets de confiance peuvent être (et sont souvent) exécutées au sein d'un gestionnaire de sécurité quelque peu indulgent mais assurant quand même un certain contrôle.

Décrire correctement un problème au client ne peut être fait uniquement par la technologie Java. Il y a plusieurs choses à considérer :

Que dire au client ?

La servlet doit-elle envoyer une page d'erreur générique correspondant au code d'état, une explication complète du problème ou (dans le cas d'une exception) un état détaillé de la pile ? Et que faire si la servlet est supposée retourner un contenu non textuel, comme une image ?

Comment enregistrer le problème ?

Doit-il être sauvé dans un fichier, écrit dans le journal du serveur, envoyé au client ou ignoré ?

Comment effectuer la reprise ?

La même instance de servlet peut-elle traiter les requêtes suivantes ? Ou bien la servlet est-elle corrompue, impliquant son rechargement ?

Les réponses à ces questions dépendent de la servlet et de son utilisation, et elles doivent être examinées au cas par cas pour chaque servlet que vous écrivez. Votre façon de gérez les erreurs vous regarde. Elle doit être basée sur le niveau de sûreté et de résistance demandé pour votre servlet. Nous allons voir à présent une vue générale des mécanismes de traitement des erreurs d'une servlet que vous pouvez utiliser pour implémenter la politique choisie.

Codes d'état

La manière la plus simple (et que l'on peut penser la meilleure) pour une servlet de rapporter une erreur est d'utiliser la méthode `sendError()` pour définir le code d'état approprié dans la série 400 ou 500. Par exemple, lorsque l'on demande à la servlet de retourner un fichier qui n'existe pas, elle peut retourner `SC_NOT_FOUND`. Quand on lui demande de faire des choses qu'elle ne sait pas faire, elle peut retourner `SC_NOT_IMPLEMENTED`. Et lorsqu'arrive ce qui n'était pas supposé arriver, elle peut retourner `SC_INTERNAL_SERVER_ERROR`.

En utilisant `sendError()` pour définir le code d'état, une servlet donne au serveur la possibilité d'effectuer un traitement spécial sur la réponse. Si l'erreur est telle que la servlet peut fournir ses propres explications au client dans le corps de la réponse, elle peut définir le code d'état avec `setStatus()` et envoyer le corps approprié, qui peut être du texte, une image, etc.

Une servlet doit faire attention à intercepter et à traiter les erreurs avant que la réponse ne soit validée. Comme vous vous en souvenez probablement (nous l'avons déjà dit plusieurs fois), HTTP spécifie que le code d'état et les en-têtes HTTP doivent être envoyés avant le corps de la réponse. Une fois le premier caractère de la réponse émis, il est trop tard pour changer le code d'état ou les en-têtes HTTP. La meilleure façon de ne pas se retrouver dans cette situation est de contrôler les erreurs et d'utiliser un tampon pour retarder l'envoi des données au client.

Configurer des pages d'erreur

Parfois, à la place d'utiliser les pages d'erreur standard du serveur, vous pourriez vouloir développer un ensemble de pages d'erreur propres à une application web et utilisées où que soit déployée cette application. Par exemple, il est possible de configurer une application web avec une page d'erreur 404 contenant un champ de saisie pour un moteur de recherche permettant à l'utilisateur de trouver la ressource « Not Found ». Vous pouvez faire cela directement en utilisant `setStatus()` et en laissant chaque servlet générer la même page d'erreur. Cependant, une meilleure approche consiste à configurer une règle `<error-page>` dans le descripteur de déploiement *web.xml* (voir l'*exemple 5-8*).

Exemple 5-8. Configurer les pages d'erreur 400 et 404

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <!-- ..... -->
    <error-page>
        <error-code>
            400
        </error-code>
        <location>
            /400.html
        </location>
    </error-page>
    <error-page>
        <error-code>
            404
        </error-code>
        <location>
            /404.html
        </location>
    </error-page>
</web-app>
```

Ces deux entrées `<error-page>` indiquent au serveur que tout appel à `sendError()` avec le code d'état 400 doit afficher le contenu de la ressource `/400.html`, le code 404 affichant la page `/404.html`. En général, les serveurs web supportent une forme de person-

nalisation de la page d'erreur similaire à celle présentée. Cependant, les entrées du fichier *web.xml* écrasent la configuration par défaut du serveur et elles offrent un mécanisme permettant à une application web de créer un ensemble standard de pages d'erreur utilisable sur toutes les implémentations de serveur. Les serveurs ont l'obligation de respecter les règles `<error-page>` pour tout le contenu provenant d'une application web, y compris les fichiers statiques.

Mais n'oubliez pas que la valeur de `<location>` doit commencer par une barre oblique, qu'elle est relative à la racine du contexte et qu'elle doit faire référence à une ressource du contexte. Pour référencer une ressource extérieure au contexte, vous pouvez pointer sur un fichier HTML du contexte qui contient une redirection immédiate sortant du contexte :

```
<META HTTP-EQUIV="Refresh" CONTENT="0; URL=http://www.errors.com/404.html">
```

La cible de `<location>` peut être une ressource dynamique, comme une servlet ou une page JSP. Pour les ressources dynamiques, le serveur fournit deux attributs de requêtes spéciaux donnant des informations sur l'erreur :

`javax.servlet.error.status_code`

: Un `Integer` donnant le code d'état de l'erreur. Le type était à l'origine non spécifié et certaines implémentations de serveur peuvent retourner le code sous forme d'un `String`.

`javax.servlet.error.message`

: Un `String` donnant le message d'état, généralement passé en second argument de la méthode `sendError()`.

Ces attributs nous permettent de créer une servlet générique affichant une page d'erreur, comme celle de l'*exemple 5-9*. Pour utiliser cette servlet, affectez-la au chemin utilisé dans la balise `<location>`.

Exemple 5-9. Créer dynamiquement une page d'erreur pour un code d'état

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ErrorDisplay extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    Object codeObj = req.getAttribute("javax.servlet.error.status_code");
    Object messageObj = req.getAttribute("javax.servlet.error.message");

    // Le code et le message ne doivent jamais être nuls dans les serveurs
    // compatibles avec l'API 2.2.
    String code = (codeObj != null ?
                   codeObj.toString() : "Code d'état manquant");
```

Exemple 5-9. Créer dynamiquement une page d'erreur pour un code d'état (suite)

```
    String message = (messageObj != null ?
                      messageObj.toString() : "Message d'erreur manquant");

    out.println("<HTML>");
    out.println("<HEAD><TITLE>" + code + " : " + message + "</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H1>" + code + "</H1>");
    out.println("<H2>" + message + "</H2>");
    out.println("<HR>");
    out.println("<I>Erreur d'accès à " + req.getRequestURI() + "</I>");
    out.println("</BODY></HTML>");
  }
}
```

Il est possible de configurer un chemin vers cette servlet comme cible de tous les codes d'erreur ayant besoin d'une page d'affichage simple de l'erreur. Une version plus sophistiquée de cette servlet pourrait générer une page plus intéressante et suivre les codes d'erreur générés afin de détecter des tendances bizarres, comme une abondance d'erreurs 404 pour le même URI, indiquant probablement la présence d'un nouveau lien erroné qu'il faut corriger.

En résumé, voici comment exploiter l'envoi des codes d'état. La méthode `setStatus()` vous permet de définir un code d'état et de garder le contrôle total sur la réponse. La méthode `sendError()` vous permet de définir un code d'état et de laisser au serveur la création de la page. Par défaut, le serveur envoie sa page d'erreur standard correspondant au code. Avec une entrée `<error-page>`, vous pouvez lui indiquer une page d'erreur spécifique.

Journalisation

Les servlets ont la possibilité d'écrire leurs actions et leurs erreurs dans un fichier de journalisation avec la méthode `log()` :

```
public void GenericServlet.log(String msg)
public void GenericServlet.log(String msg, Throwable t)
```

La méthode à un seul argument écrit le message indiqué dans un journal de servlet, habituellement un fichier de journalisation d'événements. La version à deux arguments écrit le message et la trace de la pile au moment de l'exception dans un journal de servlet. Le format de la sortie exact et l'emplacement du journal dépendent du serveur, mais le journal contient en général une date et le nom enregistré de la servlet.

La méthode `log()` aide au débogage en offrant une manière de suivre les actions d'une servlet. Elle offre aussi une manière de sauvegarder une description complète des erreurs rencontrées par la servlet. La description peut être la même que celle donnée au client, ou elle peut être plus exhaustive et détaillée.

Nous pouvons maintenant revenir en arrière et améliorer encore `ViewFile`, de façon à ce qu'elle utilise `log()` pour enregistrer sur le serveur qu'un fichier n'existe pas au lieu de retourner au client une simple page 404 Not Found :

```
// Retourner le fichier.
try {
  ServletUtils.returnFile(file, out);
}
catch (FileNotFoundException e) {
  log("Fichier non trouvé : " + e.getMessage());
  res.sendError(res.SC_NOT_FOUND);
}
```

Pour des erreurs plus complexes, une servlet peut journaliser une trace complète de la pile :

```
// Retourner le fichier.
try {
  ServletUtils.returnFile(file, out);
}
catch (FileNotFoundException e) {
  log("Fichier non trouvé : " + e.getMessage());
  res.sendError(res.SC_NOT_FOUND);
}
catch (IOException e) {
  log("Problème d'envoi du fichier", e);
  res.sendError(res.SC_INTERNAL_SERVER_ERROR);
}
```

Rapports

En plus de journaliser les erreurs et les exceptions pour l'administrateur du serveur, pendant le développement il est souvent pratique d'imprimer une description complète du problème ainsi que l'état de la pile. Malheureusement, une exception ne peut retourner la trace de la pile sous forme d'un `String`, elle ne peut que l'imprimer vers un `PrintStream` ou un `PrintWriter`. Pour l'obtenir sous forme d'un `String`, nous devons aller un peu plus avant. Nous devons faire en sorte que `Exception` imprime dans un `PrintWriter` spécial construit autour d'un `ByteArrayOutputStream`. Ce `ByteArrayOutputStream` peut intercepter la sortie et la convertir en un `String`. La classe `com.oreilly.servlet.ServletUtils` a une méthode `getStackTraceAsString()` qui fait exactement cela :

```
public static String getStackTraceAsString(Throwable t) {
  ByteArrayOutputStream bytes = new ByteArrayOutputStream();
  PrintWriter writer = new PrintWriter(bytes, true);
  t.printStackTrace(writer);
  return bytes.toString();
}
```

Voici comment `ViewFile` peut fournir une information qui inclut la pile lors d'une `IOException` :

```
// Retourner le fichier.
try {
  ServletUtils.returnFile(file, out);
}
```

```
catch (FileNotFoundException e) {
  log("Fichier non trouvé : " + e.getMessage());
  res.sendError(res.SC_NOT_FOUND);
}
catch (IOException e) {
  log("Problème d'envoi du fichier", e);
  res.sendError(res.SC_INTERNAL_SERVER_ERROR,
                ServletUtils.getStackTraceAsString(e));
}
```

La sortie pour un exemple d'exception est montrée dans la *figure 5-2*.

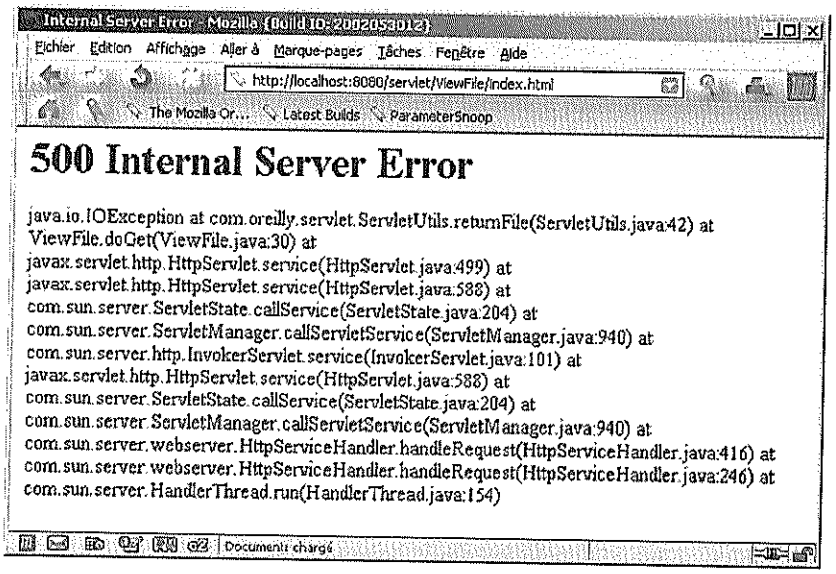


Figure 5-2. Bien informer le client

Exceptions

Comme nous l'avons dit précédemment, toute exception lancée mais non interceptée par une servlet l'est par son serveur. Le traitement de l'exception est spécifique au serveur : il peut passer un message au client et la trace de la pile, ou non. Il peut journaliser automatiquement l'exception, ou non. Il peut même appeler `destroy()` sur la servlet et la recharger, ou non.

Les servlets conçues et développées pour fonctionner avec un serveur particulier peuvent être optimisées pour le comportement de ce serveur. Une servlet conçue pour inter-opérer sur plusieurs serveurs ne peut espérer une gestion particulière des exceptions de la part du serveur. Si une telle servlet demande une gestion particulière, elle doit intercepter ses propres exceptions et les traiter de façon appropriée.

Il y a certains types d'exceptions qu'une servlet est obligée d'intercepter elle-même. Une servlet ne peut propager à son serveur que les exceptions qui sont une sous-classe de `IOException`, de `ServletException` ou de `RuntimeException`. La raison réside dans la signature des méthodes. La méthode `service()` de `Servlet` déclare dans sa clause `throws`

qu'elle lance les exceptions IOException et ServletException. Le fait de les lancer (elle ou les méthodes doGet() et doPost() qu'elle appelle) sans rien intercepter d'autre provoque une erreur de compilation. L'exception RuntimeException est une exception spéciale qui n'a jamais besoin d'être déclarée dans une clause throws. Un exemple classique est l'exception NullPointerException.

Les méthodes init() et destroy() ont aussi leur propre signature. La méthode init() déclare qu'elle ne lance que les exceptions ServletException, et destroy() déclare qu'elle ne lance aucune exception.

ServletException

ServletException est une sous-classe de java.lang.Exception spécifique aux servlets ; la classe est définie dans le paquetage javax.servlet. Cette exception est lancée pour indiquer un problème général dans la servlet. Elle a les mêmes constructeurs que java.lang.Exception : un qui ne prend pas d'arguments et un qui prend un seul message sous forme de chaîne de caractères :

```
javax.servlet.ServletException()
javax.servlet.ServletException(String msg)
```

ServletException peut aussi supporter un objet Throwable servant de « cause racine ». Cela permet à ServletException de se comporter comme une enveloppe autour de n'importe quel type d'exception ou d'erreur. Ainsi, le serveur dispose d'un moyen de connaître le problème « racine » ayant provoqué le lancement de l'exception ServletException. Pour cela, ServletException a deux constructeurs supplémentaires :

```
javax.servlet.ServletException(Throwable rootCause)
javax.servlet.ServletException(String msg, Throwable rootCause)
```

Grâce à la possibilité de cause racine, vous pouvez passer une exception ou une erreur sous-jacentes :

```
try {
  thread.sleep(60000);
}
catch (InterruptedException e) {
  throw new ServletException(e);  // Inclure l'exception sous-jacente.
}
```

Le serveur peut retrouver et examiner l'exception sous-jacente, la trace de la pile et tout le reste, en appelant getRootCause() :

```
public Throwable ServletException.getRootCause()
```

L'appel retourne null s'il n'y a aucune exception imbriquée.

UnavailableException

Le paquetage javax.servlet définit une sous-classe de ServletException, UnavailableException, bien que vous puissiez ajouter la vôtre. Cette exception indique qu'une servlet est indisponible, soit temporairement soit de façon permanente.

Une indisponibilité permanente signifie que l'instance de la servlet lançant l'exception UnavailableException ne peut résoudre une erreur. La servlet est peut-être mal configurée ou son état peut être corrompu.

Une servlet qui lance une exception `UnavailableException` permanente pendant la gestion d'une requête est retirée du service et une nouvelle instance est créée afin de traiter les requêtes. Si aucune instance de servlet « disponible » ne peut être créée, le client reçoit une erreur. Une servlet qui lance une exception `UnavailableException` permanente (ou une `ServletException` normale) dans sa méthode `init()` n'entre jamais en service ; à la place, le serveur essaie d'initialiser une nouvelle instance pour gérer les requêtes suivantes.

Une indisponibilité temporaire signifie que la servlet ne peut gérer les requêtes pendant un certain temps du fait d'un problème système. Par exemple, un serveur trois-tiers peut ne pas être accessible ou il peut manquer de ressources mémoire ou disque pour traiter les requêtes. Le problème peut se corriger tout seul, par exemple s'il s'agit d'une charge excessive, ou un administrateur a peut-être besoin d'intervenir.

Pendant l'indisponibilité, le serveur traite les requêtes pour la servlet en renvoyant le code d'état `SC_SERVICE_UNAVAILABLE` (503) accompagné d'un en-tête `Retry-After` indiquant au client la date de fin estimée de l'indisponibilité. Si une servlet lance une exception `UnavailableException` temporaire pendant sa méthode `init()`, elle n'entre jamais en service et le serveur essaie d'initialiser une nouvelle instance après la période d'indisponibilité. Pour des raisons de simplicité, les serveurs sont autorisés à traiter l'indisponibilité temporaire comme une l'indisponibilité permanente.

`UnavailableException` a deux constructeurs :

```
javax.servlet.UnavailableException(String msg)
javax.servlet.UnavailableException(String msg, int seconds)
```

Le constructeur à un argument crée une nouvelle exception indiquant que la servlet donnée est indisponible de façon permanente, avec une explication donnée par `msg`. La version à deux arguments crée une nouvelle exception indiquant que la servlet donnée est indisponible temporairement, avec une explication donnée par `msg`. Une servlet bien écrite doit inclure dans l'explication la raison du problème et toute action corrective que l'administrateur système doit effectuer afin que la servlet redevienne disponible. La durée de l'indisponibilité est donnée en secondes. Ce temps n'est qu'une estimation. Si une estimation ne peut être faite, une valeur négative peut être utilisée. `UnavailableException` fournit les méthodes `isPermanent()` et `getUnavailableSeconds()` pour retrouver les informations sur l'exception.

Configurer les pages d'exception

Le comportement d'un serveur quand il intercepte une exception de servlet dépend du serveur. Cependant, une application web peut indiquer, *via* son descripteur de déploiement, un ensemble de pages d'erreur pour traiter certaines exceptions. Les pages d'erreur sont spécifiées à l'aide de la balise `<error-page>`, comme dans le cas des codes d'état, mais `<error-code>` est remplacé par `<exception-type>`. Voir l'*exemple 5-10*.

Exemple 5-10. Configurer les pages d'erreurs pour les exceptions

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
```

Exemple 5-10. Configurer les pages d'erreurs pour les exceptions (suite)

```
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <!-- ..... -->
    <error-page>
        <exception-type>
            javax.servlet.ServletException
        </exception-type>
        <location>
            /servlet/ErrorDisplay
        </location>
    </error-page>
</web-app>
```

Cette entrée indique qu'une exception `ServletException` lancée au serveur doit être affichée à l'aide de la servlet `ErrorDisplay`. Notez que `<exception-type>` doit être complètement qualifié avec le nom de paquetage ; `ServletException` seul ne peut fonctionner. Cette règle s'applique aussi à toutes les sous-classe de `ServletException` comme `UnavailableException`, sauf si le descripteur de déploiement contient une règle plus spécifique pour `UnavailableException`.

Pour les cibles `<location>` dynamiques, le serveur fournit deux attributs de requêtes décrivant l'exception lancée :

`javax.servlet.error.exception_type`

Une instance de `java.lang.Class` indiquant le type de l'exception. Cet attribut était à l'origine non spécifié et certaines implémentations de serveur peuvent renvoyer un nom de classe sous forme de `String`.

`javax.servlet.error.message`

Un `String` donnant le message de l'exception, passé dans le constructeur de l'exception. Il n'existe aucun moyen d'obtenir l'exception ou sa trace de la pile.

Avec ces attributs, nous pouvons améliorer `ErrorDisplay` de l'*exemple 5-9* afin qu'elle fonctionne comme une ressource générale pour l'affichage des erreurs, acceptant l'affichage des codes d'état et des exceptions (voir l'*exemple 5-11*).

Exemple 5-11. Créer dynamiquement une page d'erreur générique

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ErrorDisplay extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
```

Exemple 5-11. Créer dynamiquement une page d'erreur générique (suite)

```
        String code = null, message = null, type = null;
        Object codeObj, messageObj, typeObj;

        // Retrouver les trois attributs d'erreur possibles, certains peuvent
        // être nuls.
        codeObj = req.getAttribute("javax.servlet.error.status_code");
        messageObj = req.getAttribute("javax.servlet.error.message");
        typeObj = req.getAttribute("javax.servlet.error.exception_type");

        // Convertir les attributs en String.
        if (codeObj != null) code = codeObj.toString();
        if (messageObj != null) message = messageObj.toString();
        if (typeObj != null) type = typeObj.toString();

        // La raison de l'erreur est soit le code d'état, soit le type
        // de l'exception.
        String reason = (code != null ? code : type);

        out.println("<HTML>");
        out.println("<HEAD><TITLE>"+reason+" : " + message + "</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<H1>" + reason + "</H1>");
        out.println("<H2>" + message + "</H2>");
        out.println("<HR>");
        out.println("<I>Erreur d'accès à " + req.getRequestURI() + "</I>");
        out.println("</BODY></HTML>");
    }
}
```

Malheureusement, seuls le type et le message de l'exception, non la trace de la pile, sont disponibles à la servlet `ErrorDisplay`. L'utilité de la page d'erreur pour l'exception s'en trouve donc limitée. Pour garantir l'affichage ou la journalisation de la trace de la pile, vous devez intercepter l'exception dans la servlet d'origine et la traiter avant de la propager au serveur. Vous devez donc écrire un peu plus de code, mais votre propre gestion des exceptions peut garantir une gestion des erreurs cohérente et adaptée. L'API Servlet 2.3 devrait inclure un nouvel attribut de requête contenant l'exception elle-même, sous le nom `javax.servlet.error.exception`.

Savoir quand plus personne n'écoute

Parfois, les client se déconnectent des servlets. C'est vexant, mais ça arrive. Quelquefois, le client fait une erreur et va à la mauvaise page. Quelquefois, la servlet est trop longue à répondre. Souvenez-vous que pendant qu'une servlet prépare sa réponse, l'utilisateur est tenté de presser le gros bouton du navigateur marqué Stop. Vous pourriez vous demander ce qui arrive à la servlet lorsque ce bouton a été pressé.

Malheureusement, une servlet n'est pas immédiatement informée que l'utilisateur a pressé le bouton Stop ; il n'y a pas d'interruption lui indiquant d'arrêter son travail. La servlet découvre que le client a arrêté la requête uniquement lorsqu'elle essaie d'en-

voyer la sortie au client disparu, c'est à ce moment-là qu'une condition d'erreur survient.

Une servlet qui envoie des informations en utilisant `ServletOutputStream` voit une exception `IOException` quand elle essaie d'écrire sa sortie. Pour les serveurs qui mettent en tampon la sortie, l'`IOException` est lancée quand le tampon déborde et que son contenu est vidé.

Une exception `IOException` pouvant être lancée à tout moment dès qu'une servlet essaie d'écrire sa sortie, une servlet bien programmée libérera ses ressources dans un bloc `finally`. (Le bloc `finally` est une partie optionnelle d'une construction `try/catch/finally`. Il vient après zéro ou plusieurs blocs `catch` et son code est exécuté une fois indépendamment de l'exécution du code du bloc `try`.) Voici une version de la méthode `returnFile()` de la servlet `ViewFile` qui utilise un bloc `finally` pour garantir la fermeture de son `FileInputStream` :

```
void returnFile(String filename, OutputStream out)
                        throws FileNotFoundException, IOException {
  FileInputStream fis = null;
  try {
    fis = new FileInputStream(filename);
    byte[] buf = new byte[4 * 1024];  // Tampon de 4K.
    int bytesRead;
    while ((bytesRead = fis.read(buf)) != -1) {
      out.write(buf, 0, bytesRead);
    }
  }
  finally {
    if (fis != null) fis.close();
  }
}
```

L'ajout du bloc `finally` ne change pas le fait que cette méthode propage toutes les exceptions à son appelant, mais il garantit que, avant cette propagation, la méthode a une chance de fermer le `FileInputStream` ouvert.

Une servlet envoyant des données caractères en utilisant un `PrintWriter` ne reçoit pas une exception `IOException` lorsqu'elle essaie d'écrire sa sortie, car les méthodes de `PrintWriter` ne lancent jamais d'exceptions. À la place, une servlet qui envoie des données caractères doit appeler la méthode `checkError()` de `PrintWriter`. Cette méthode vide la sortie, validant la réponse, et retourne un `boolean` qui indique s'il y a eu un problème pendant l'écriture dans le `OutputStream` sous-jacent. Elle retourne `true` si le client a arrêté la requête.

Une servlet longue qui ne pense pas à valider sa réponse doit appeler `checkError()` régulièrement afin de déterminer si elle doit arrêter son travail plus tôt. S'il n'y a pas eu de sortie depuis la dernière vérification, une servlet peut envoyer un contenu de remplissage. Par exemple :

```
out.println("<H2>Voici la solution pour votre équation " +
            "différentielle :</H2>");
if (out.checkError()) return;
```

```
preliminaryCalculation();

out.print(" "); // Contenu de remplissage, les espaces supplémentaires
                // sont ignorés en HTML.
if (out.checkError()) return;

additionalCalculation();
```

Il est important de noter qu'un serveur n'est pas obligé de lancer une `IOException` ou de positionner l'indicateur d'erreur de `PrintWriter` après la déconnexion du client. Un serveur peut choisir de laisser la réponse arriver à sa fin en ignorant sa sortie. Généralement, ce n'est pas un problème, mais cela veut dire qu'une servlet s'exécutant dans un tel serveur doit toujours avoir défini un point d'arrêt et ne doit pas être écrite pour boucler continuellement jusqu'à ce que l'utilisateur appuie sur Stop.

Six façons d'habiller une servlet

Depuis la première édition de ce livre, les servlets sont devenues le standard *de facto* pour le développement Java côté serveur web. La bagarre Darwinienne entre les servlets, les applets côté serveur et les autres architectures Java enfichables (*plug-in*) s'est effectivement terminée et les servlets ont été déclarées vainqueur. Elles sont aujourd'hui supportées par tout serveur web et tout serveur d'applications. Le nouveau domaine d'innovation intense se trouve au-dessus de la couche des servlets, au niveau de la présentation et du framework, là où des personnes et des sociétés cherchent comment tirer au mieux parti des servlets dans la création de sites web.

Ces solutions sont nécessaires car l'approche consistant à générer du contenu par des appels à `out.println()` a prouvé son inaptitude dans une utilisation réelle. Avec cette approche, le contenu balisé doit être créé dans le code et cela devient une tâche longue et onéreuse pour les pages importantes. De plus, les créateurs de pages doivent demander aux développeurs d'effectuer toutes les modifications du site.

L'objectif de quasiment toutes les architectures de création de contenu est de séparer le contenu de la présentation. Ici, le termine *contenu* signifie les données brutes du site et la manipulation de ces données. Un meilleur terme serait peut-être *traitement des données*, mais séparer le traitement des données de la présentation ne semble pas être naturel. Le terme *présentation* signifie la représentation des données à l'utilisateur final (souvent du HTML, même si avec l'augmentation des périphériques connectés au Web, ce sera de plus en plus du WML ou *Wireless Markup Language*). La séparation du contenu de la présentation présente plusieurs avantages, et tout d'abord une création et une maintenance du site plus faciles car le *contenu* de la page (contrôlé par un programmeur) peut être développé et modifié indépendamment de la *présentation* de la page (contrôlée par un designer ou un producteur). Certains comparent cette séparation à une architecture Modèle-Vue-Contrôleur (MVC). Le *modèle* est synonyme de *contenu*, la *vue* est synonyme de *présentation*, et le *contrôleur* de trouve à différent endroits en fonction de la technologie.

Plus loin, nous verrons en détails un certain nombre d'alternatives à la création de contenu basée sur les servlets. Pour chaque solution, nous donnerons des informations sur l'outil, nous montrerons comment l'utiliser et nous examinerons les cas les mieux adap-

tés à l'outil. Pour des raisons de temps et de place, ainsi que le fait que chaque projet mentionné évolue rapidement, nous ne pouvons pas donner pour chacun un tutoriel complet. Mais que ce soit bien clair, il existe plus de cinq implémentations différentes de la même idée. Chaque solution approche le problème de création de contenu sous un angle différent et conduit donc à des techniques de résolution du problème très différentes. Un outil qui fonctionne bien dans un cas peut ne pas être adapté à une autre situation. Ne lisez donc pas les exemples de ces chapitres à la recherche de la meilleure technologie. Recherchez plutôt la technologie la mieux adaptée à votre projet.

Nous avons rejeté une présentation des solutions commerciales propriétaires du fait du risque inhérent de lien à une solution propre à un fournisseur créé par des motivations financières et commerciales.[1] Les solutions présentées plus loin sont les suivantes :

JavaServer Pages

Les JSP (*JavaServer Pages*) sont une technologie créée par Sun Microsystems et intimement liée aux servlets. Comme pour les servlets, Sun a publié une spécification des JSP et des fournisseurs tiers mettent en concurrence leur implémentation de cette norme. En étant publiées par Sun, les JSP sont dans une position très privilégiée et si les JSP avaient résolu un nombre suffisamment important de problèmes des utilisateurs, elles auraient certainement gagné le marché avant que d'autres solutions viables existent. En fait, un nombre surprenant d'utilisateurs sont désenchantés par les JSP et les autres approches gagnent en popularité. Voir le chapitre 18, *JavaServer Pages*, et *http://java.sun.com/products/jsp*.

Tea

Tea est un nouveau produit en *open source* du Walt Disney Internet Group (précédemment GO.com) créé en interne ces dernières années afin de résoudre les besoins énormes de la production des sites web tels que ESPN.com. Tea est similaire aux JSP tout en évitant beaucoup de leurs problèmes. Il possède déjà un grand nombre d'outils. Voir le chapitre 14, *L'environnement Tea*, et *http://opensource.go.com*.

WebMacro

WebMacro est un moteur de template créé par Semiotek, Inc., pour le projet Shimari. Il sera peut-être inclus dans Apache Jakarta Project. On pourrait présenter beaucoup de moteurs de template. Cependant, WebMacro est le plus utilisé, ce dans des sites commerciaux à très fort trafic comme AltaVista.com, il a été intégré à des frameworks en *open source* comme Turbine et Melati, et il a été utilisé dans des gros projets en *open source* comme JetSpeed. Voir le chapitre 15, *WebMacro*, *http://webmacro.org* et *http://jakarta.apache.org/velocity*.

---

1. Nous l'avons appris à nos dépend. Dans la première édition de ce livre, nous décrivions le paquetage commercial htmlKona de WebLogic car il n'existait pas de solution similaire et parce que WebLogic nous avait assuré que son prix resterait abordable. Après le rachat de WebLogic par BEA Systems, cette garantie n'était plus de mise et, en fait, htmlKona est devenu une note de base de page dans le catalogue de BEA Systems. Heureusement, htmlKona a été réimplémenté en *open source* au sein du projet ECS (*Element Construction Set*) par le biais du Apache Jakarta Project. Il sera maintenant disponible et maintenu tant que des personnes seront intéressées par ce projet.

Element Construction Set

Le paquetage ECS (*Element Construction Set*) du Apache Jakarta Project est un ensemble de classes modélisées d'après le produit htmlKona de WebLogic (maintenant BEA Systems). ECS a beaucoup de limitations, mais il résout une certaine classe de problèmes et un examen de l'approche ECS donne de bonnes bases de discussion pour XMLC qui est plus souple. Voir le chapitre 16, *Element Construction Set*, et *http://jakarta.apache.org/ecs*.

XMLC

XMLC utilise XML pour exploiter toute la puissance d'ECS sans en avoir les limitations. Il a été créé par Lutris dans leur projet en *open source* Enhydra Application Server et il peut être utilisé comme un composant séparé. Voir le chapitre 17, *XMLC*, et *http://xmlc.enhydra.org*.

Cocoon

Cocoon est une autre solution basée sur XML créée par le projet XML Apache. Cocoon utilise XML et XSLT pour gérer le contenu et créer ce que l'on a souvent appelé un *framework de publication web*. Cocoon est un framework reposant sur les servlets et dans lequel le contenu XML, statique ou dynamique, est passé par un filtre XSLT avant d'être présenté au client. Ce filtre peut formater le contenu dans différents types, dont HTML, XML, WML ou PDF. Le framework peut choisir différents types suivant le client. Nous ne présenterons pas en détail Cocoon dans ce livre car il est plus en rapport avec la programmation XSLT qu'avec la programmation Java et parce qu'il n'est pas généralement (au moins jusqu'à présent) l'outil adapté à l'écriture d'applications web interactives. Il est mieux adapté à des sites importants principalement statiques et qui veulent avoir des vues multiples de leur contenu. Pour plus d'informations sur Cocoon, consultez *http://xml.apache.org* et le livre *Java et XML* de Brett McLaughlin (Éditions O'Reilly). Le chapitre de *Java et XML* qui présente Cocoon est disponible gratuitement en ligne à *http://www.oreilly.com/catalog/javaxml/chapter/ch09.html*.

Ne soyez pas surpris si votre outil favori ou un outil populaire n'est pas mentionné. Regardez la date de création de votre outil : elle est probablement postérieure à la date d'impression de ce livre. L'innovation dans ce domaine est permanente, et les nouvelles possibilités apportées par XML et les périphériques web demandant de nouvelles fonctionnalités, ce domaine est mûr pour de nouvelles entrées. Comprenez aussi que les améliorations de ces outils arrivent à une vitesse folle. Chaque chapitre les décrivant est inévitablement un peu en retard au moment où vous le lisez. Pour restez à jour, consultez *http://www.servlets.com*.

6

Envoyer un contenu multimédia

Jusqu'à présent, les servlets que nous avons écrites ont retourné une page HTML standard. Mais le Web ne se limite pas à HTML. Dans ce chapitre nous verrons que les servlets peuvent retourner des choses plus intéressantes. Nous commencerons par le nouveau protocole WAP et le langage WML utilisés par les téléphones et les périphériques mobiles et nous verrons comment les servlets utilisent cette nouvelle technologie. Ensuite, nous verrons comment générer dynamiquement des images dans une servlet pour créer des graphes et pour manipuler des images. Vers la fin de ce chapitre, nous verrons aussi quand et comment envoyer une réponse compressée et nous examinerons les réponses multipartie servant à implémenter le push du serveur.

WAP et WML

WAP (*Wireless Application Protocol*) est un standard *de facto* pour l'implantation des communications Internet dans les téléphones mobiles, les pagers et les assistants personnels numériques (PDA — *personal digital assistants*) exploitant les réseaux sans fil du monde. Il a été créé en 1998 par Ericsson, Nokia, Motorola et Phone.com (précédemment Unwired Planet) qui avaient prudemment souhaité développer un protocole standard plutôt que des protocoles propriétaires concurrents. Ces quatre entreprises ont constitué le WAP Forum (*http://www.wapforum.org*), un groupe démocratique dont le nombre de membres a depuis dépassé 400.

WAP repose sur un ensemble de spécifications pour le développement d'applications qui s'exécutent sur des réseaux sans fil. Le protocole WAP concerne le niveau application (le langage à balises WML et le langage de script WMLScript, connu sous le nom WAE pour *Web Application Environment*) et les couches de transport réseau sous-jacentes (les protocoles WDP, WTLS, WTP et WSP). La pile WAP reproduit la pile de protocole du Web (voir la *figure 6-1*).

La principale différence réside dans le fait que WAP est optimisé pour les communications sans fil à faible bande passante. Par exemple, WAP utilise un format binaire pour la structure des requêtes et des réponses au lieu du format textuel du Web.

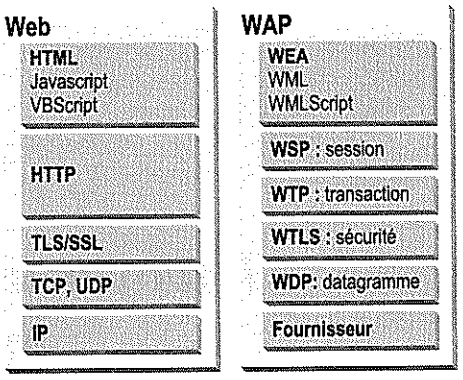


Figure 6-1. Piles Web et WAP

Une *passerelle WAP* agit comme un intermédiaire entre le réseau WAP et le Web. Une passerelle convertit une requête WAP en une requête web, puis la réponse web en une réponse WAP. Vous pouvez voir une passerelle WAP comme un « convertisseur de pile de protocoles »[1] Les passerelles WAP apportent aux périphériques WAP un accès aux serveurs web standard. Voir la *figure 6-2.*

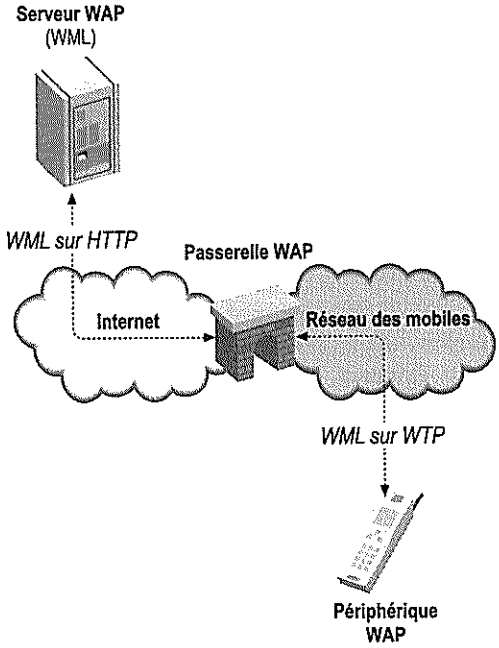


Figure 6-2. Rôle d'une passerelle WAP

1. Les passerelles WAP sont normalement apportées par le fournisseur de réseau sans fil. Pour ceux qui s'intéressent à la configuration de leur propre passerelle ou à son fonctionnement, il existe une passerelle WAP en *open source* appelée Kannel et disponible à *http://www.wapgateway.org.*

En fait, en tant que développeur pour le sans-fil, vous pouvez ignorer la couche de transport WAP. Une passerelle WAP prend en charge la conversion d'une requête d'un périphérique en une requête HTTP. La couche qui vous concerne est la couche application, où, à la place de générer du HTML, vous générez du WML.

WML

Les périphériques mobiles sont bien plus limités que les PC. Ils ont des processeurs lents, de faibles quantités de mémoire, de petits écrans et une bande passante très limitée. Du fait de ces limitations, les périphériques WAP n'interagissent pas avec du contenu HTML et des images normaux. À la place, ils utilisent WML (*Wireless Markup Language*) pour le contenu textuel, WMLScript pour les scripts et WBMP (*Wireless Bitmap*), un format d'images monochromes, pour le graphisme.

WML est une application de XML, similaire à HTML mais avec bien moins de balises. Il suit la métaphore du jeu de cartes. Chaque carte représente un écran ou une page, et chaque jeu est un ensemble de cartes pouvant être transféré en une fois. L'envoi du contenu un jeu à la fois au lieu d'une carte à la fois réduit la latence (le temps pris pour voir la carte suivante) lors des accès au contenu.

L'*exemple 6-1* montre un document WML statique agissant comme une mini-carte des consommations. Il fournit une liste de boissons et affiche les ingrédients de celle sélectionnée.[2] Notez que le document est en XML avec une DTD bien connue.

Exemple 6-1. Une mini-carte des consommations WML, drinks.wml

```
<?xml version="1.0"?>

<!DOCTYPE wml PUBLIC
  "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
  <card id="Bartender" title="Choisissez une boisson">
    <p>
    Choisissez une boisson :
    <anchor>
      Kamikaze <go href="#Kamikaze" />
    </anchor><br/>
    <anchor>
      Margarita <go href="#Margarita" />
    </anchor><br/>
    <anchor>
      Boilermaker <go href="#Boilermaker" />
    </anchor><br/>
    </p>
  </card>
```

2. Une implémentation complète est publiquement disponible à *http://www.wap2bar.com/index.wml*.

Exemple 6-1. Une mini-carte des consommations WML, drinks.wml (suite)

```
  <card id="Kamikaze" title="Kamikaze">
    <p>
    Pour faire un Kamikaze :<br/>
    6 cl de Vodka<br/>
    0,5 cl de Triple Sec<br/>
    0,5 cl de jus de citron<br/>
    </p>
  </card>

  <card id="Margarita" title="Margarita">
    <p>
    Pour faire une Margarita :<br/>
    4 cl de Tequila<br/>
    2 cl de Triple Sec<br/>
    1 cl de Jus de citron vert<br/>
    Sel<br/>
    </p>
  </card>

  <card id="Boilermaker" title="Boilermaker">
    <p>
    Pour faire un Boilermaker :<br/>
    4,5 cl de Whiskey<br/>
    25 cl de Bière<br/>
    </p>
  </card>
</wml>
```

Le document contient quatre cartes. La première, affichée par défaut, donne une courte liste de boissons. Chaque nom de boisson est un lien vers une autre carte du document, le nom de la carte suivant la syntaxe `#NomDeLaCarte`. Les dernières cartes affichent les ingrédients d'une boisson donnée. Le document complet peut être transféré sur un périphérique en une seule étape, même si seules certaines parties sont disponibles à un instant donné.

Simulateurs de périphérique WAP

Pour pouvoir exécuter des applications WAP sans passer par les ondes, plusieurs sociétés ont créé des simulateurs de téléphones compatibles WAP. Ce sont des logiciels qui s'exécutent sur un PC et qui agissent (et même ressemblent !) à un téléphone. La *figure 6-3* montre le document tel qu'il apparaît dans l'UP.Simulator de Phone.com (*http://www.phone.com*). D'autres simulateurs et des kits de développement WAP sont disponible sur les sites *http://www.nokia.com*, *http://www.ericsson.com*, *http://www.motorola.com*, et d'autres. Certains navigateurs web standard, dont Opera, acceptent aussi WML.

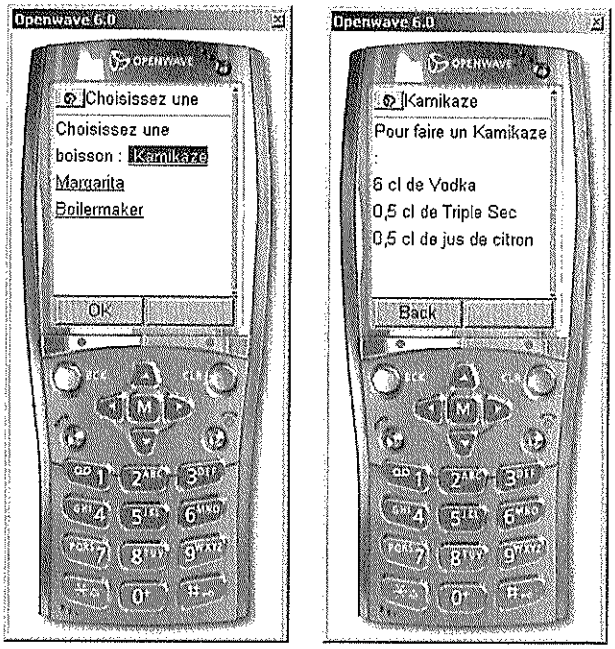


Figure 6-3. Boire ou téléphoner

Envoyer du contenu WAP

Pour que le contenu WAP soit traité correctement, le serveur doit envoyer le fichier WML avec le type de contenu `text/vnd.wap.wml`. Certains serveurs emploient ce type de contenu automatiquement pour les fichiers *.wml*. D'autres doivent être informés du nouveau type. Pour informer un serveur des types de contenu liés au WAP, il est possible d'utiliser les éléments `<mime-mapping>` suivants dans son descripteur de déploiement *web.xml*. L'extrait de *web.xml* de l'*exemple 6-2* ajoute les types de contenu adaptés à WML, WMLScript et aux fichiers WBMP.

Exemple 6-2. Ajouter les types MIME WAP à web.xml

```
<!-- ... -->
<mime-mapping>
    <extension>
        wml
    </extension>
    <mime-type>
        text/vnd.wap.wml
    </mime-type>
</mime-mapping>
<mime-mapping>
    <extension>
        wmls
    </extension>
```

Exemple 6-2. Ajouter les types MIME WAP à web.xml (suite)

```
    <mime-type>
        text/vnd.wap.wmlscript
    </mime-type>
</mime-mapping>
<mime-mapping>
    <extension>
        wbmp
    </extension>
    <mime-type>
        image/vnd.wap.wbmp
    </mime-type>
</mime-mapping>
<!-- ... -->
```

De plus, certains programmeurs d'applications web jugent utile d'ajouter *index.wml* à la liste par défaut des fichiers à rechercher dans un répertoire, connu comme étant la *liste des fichiers d'accueil*. L'extrait du descripteur de déploiement *web.xml* de l'*exemple 6-3* définit la liste des fichiers d'accueil à *index.html*, *index.htm* et *index.wml*, dans cet ordre. Cela indique au serveur qu'une requête sur *http://localhost/wap* doit correspondre à *http://localhost/wap/index.wml* si *index.html* et *index.htm* n'existent pas.

Exemple 6-3. Ajouter les fichiers d'accueil WAP à web.xml

```
<!-- ... -->
<welcome-file-list>
  <welcome-file>
    index.html
  </welcome-file>
  <welcome-file>
    index.htm
  </welcome-file>
  <welcome-file>
    index.wml
  </welcome-file>
</welcome-file-list>
<!-- ... -->
```

La fonctionnalité `<welcome-file-list>` est aussi intéressante pour les utilisations non-WAP. Par exemple, une application web utilisant, pour des raisons historiques, *home.html* comme fichier d'accueil par défaut peut ajouter *home.html* à la liste des fichiers d'accueil et on peut alors la déployer de façon sûre sur tout serveur compatible avec les servlets.

Contenu WAP dynamique

Pour une servlet, il n'y a presqu'aucune différence technique entre l'envoi d'un contenu WAP dynamique et un contenu web dynamique. Pour servir du WAP, une servlet doit simplement mettre le type du contenu de la réponse à `text/vnd.wap.wml` et remplacer le HTML de la réponse par du WML. Pour la servlet, la requête est une requête HTTP

normale ; la passerelle WAP externe gère la conversion entre les protocoles WAP et web. Mais, bien que la différence technique soit minime, les occasions d'utiliser une servlet pour générer du contenu dynamique sont souvent plus rares.

Sur le Web, il existe généralement peu de diminution des performances quand on contacte un serveur pour des tâches mineures et des mises à jour de pages. Avec le WAP, l'impact sur les performances est bien plus notable. Par conséquent, de simples tâches comme le passage d'une carte à l'autre ou la validation des données d'un formulaire, qui, sur le Web, peuvent impliquer une servlet, sont bien plus efficaces sur le WAP quand elles sont effectuées à l'aide de la métaphore du jeu de cartes de WML et des possibilités de scripts côté client apportées par WMLScript. Par exemple, la carte des consommations précédente utilise un ensemble de cartes WML pour présenter ses boissons sans contacter le serveur.

Néanmoins, les servlets ont encore leur place. Une servlet (ou tout autre script côté serveur) est nécessaire pour la génération d'un jeu de cartes contenant des informations dynamiques extraites d'une base de données. Une servlet est aussi nécessaire pour gérer les requêtes sur des quantités de donnée trop importantes pour tenir dans un ensemble de cartes. Les périphériques sont souvent limités au stockage de 1 400 octets de données pour des pages compilées.

L'*exemple 6-4* et l'*exemple 6-5* montrent un formulaire WML et une servlet générant du WML qui fournissent une application de recherche d'indicatif régional. Un client peut entrer un indicatif régional téléphonique dans le formulaire, l'envoyer à la servlet et savoir à quelle région correspond ce code. Les personnes disposant d'un téléphone compatible WAP peuvent utiliser cette application pour localiser physiquement tout numéro d'appelant.

Exemple 6-4. Utiliser WML pour demander un indicatif régional

```
<?xml version="1.0"?>

<!DOCTYPE wml PUBLIC
  "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
  <card id="AreaCode" title="Entrez un indicatif régional">
    <do type="accept" label="Enter">
      <go href="servlet/AreaCode?code=$(code)"/>
    </do>
    <p>
    Entrez un indicatif régional : <input type="text" name="code"/>
    </p>
  </card>
</wml>
```

Ce document WML contient un formulaire simple avec un champ de saisie de texte. Quel que soit l'indicatif entré, il est envoyé à la servlet `AreaCode` dans le paramètre `code`. Nous utilisons la syntaxe de substitution de variable `$(code)` pour construire manuellement la chaîne d'interrogation.

Exemple 6-5. Utiliser WAP pour localiser un indicatif régional

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AreaCode extends HttpServlet {

  Properties lookup = new Properties();

  public void init() {
    // Transférer les données brutes dans une liste de propriétés à
    // recherche rapide.
    for (int i = 0; i < data.length; i++) {
      Object[] record = data[i];
      String state = (String) record[0];
      int[] codes = (int[]) record[1];
      for (int j = 0; j < codes.length; j++) {
        lookup.put(String.valueOf(codes[j]), state);
      }
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/vnd.wap.wml");
    PrintWriter out = res.getWriter();

    String msg = null;

    String code = req.getParameter("code");
    String region = null;
    if (code != null) {
      region = lookup.getProperty(code);
    }

    out.println("<?xml version=\"1.0\"?>");
    out.println("<!DOCTYPE wml PUBLIC " +
                "\"-//WAPFORUM//DTD WML 1.1//EN\" " +
                "\"http://www.wapforum.org/DTD/wml_1.1.xml\">");

    out.println("<wml>");
    out.println("<card id=\"Code\" title=\"Code\">");
    out.println("  <p>");
    out.println("  L'indicatif régional '" + code + "'<br/>");
    if (region != null) {
      out.println("  est pour " + region + ".<br/>");
    } else {
      out.println("  n'est pas valide.<br/>");
    }
```

Exemple 6-5. Utiliser WAP pour localiser un indicatif régional (suite)

```
    out.println("  </p>");
    out.println("</card>");
    out.println("</wml>");
  }

  // Indicatifs de chaque région.
  private Object[][] data = new Object[][] {
    { "les appels gartuits", new int[] { 800, 855, 866, 877, 888 } },
    { "l'Alabama", new int[] { 205, 256, 334 } },
    { "l'Alaska", new int[] { 907 } },
    { "l'Alberta", new int[] { 403, 780 } },
    { "l'Arizona", new int[] { 480, 520, 602, 623 } },
    { "l'Arkansas", new int[] { 501, 870 } },
    { "la Colombie britanique", new int[] { 250, 604 } },
    { "la Californie", new int[] { 209, 213, 310, 323, 369, 408, 415, 424,
      510, 530, 559, 562, 619, 626, 627, 650, 661, 707, 714, 760, 805, 818,
      831, 858, 909, 916, 925, 949 } },
    { "le Colorado", new int[] { 303, 719, 720, 970 } },
    { "le Connecticut", new int[] { 203, 475, 860, 959 } },
    { "le Deleware", new int[] { 302 } },
    { "le District de Columbie", new int[] { 202 } },
    { "la Floride", new int[] { 305, 321, 352, 407, 561, 727, 786, 813, 850,
      863, 904, 941, 954 } },
    { "la Géorgie", new int[] { 229, 404, 478, 678, 706, 770, 912 } },
    { "Hawaii", new int[] { 808 } },
    { "l'Idaho", new int[] { 208 } },
    { "l'Illinois", new int[] { 217, 224, 309, 312, 618, 630, 708, 773, 815,
      847 } },
    { "l'Indiana", new int[] { 219, 317, 765, 812 } },
    { "l'Iowa", new int[] { 319, 515, 712 } },
    { "le Kansas", new int[] { 316, 785, 913 } },
    { "le Kentucky", new int[] { 270, 502, 606, 859 } },
    { "la Louisianne", new int[] { 225, 318, 337, 504 } },
    { "le Maine", new int[] { 207 } },
    { "le Manitoba", new int[] { 204 } },
    { "le Maryland", new int[] { 240, 301, 410, 443 } },
    { "le Massachusetts", new int[] { 413, 508, 617, 781, 978 } },
    { "le Michigan", new int[] { 231, 248, 313, 517, 586, 616, 734, 810,
      906 } },
    { "le Minnesota", new int[] { 218, 320, 507, 612, 651, 763, 952 } },
    { "le Mississippi", new int[] { 228, 601, 662 } },
    { "le Missouri", new int[] { 314, 417, 573, 636, 660, 816 } },
    { "le Montana", new int[] { 406 } },
    { "le Nebraska", new int[] { 308, 402 } },
    { "le Nevada", new int[] { 702, 775 } },
    { "le New Brunswick", new int[] { 506 } },
    { "le New Hampshire", new int[] { 603 } },
    { "le New Jersey", new int[] { 201, 609, 732, 856, 908, 973 } },
    { "le Nouveau Mexique", new int[] { 505 } },
```

Exemple 6-5. Utiliser WAP pour localiser un indicatif régional (suite)

```
    { "New York", new int[] { 212, 315, 347, 516, 518, 607, 631, 646, 716,
      718, 845, 914, 917 } },
    { "le Newfoundland", new int[] { 709 } },
    { "la Caroline du Nord", new int[] { 252, 336, 704, 828, 910, 919, 980 } },
    { "le Dakota du Nord", new int[] { 701 } },
    { "les Territoires du Nord-Ouest", new int[] { 867 } },
    { "le Nova Scotia", new int[] { 902 } },
    { "l'Ohio", new int[] { 216, 234, 330, 419, 440, 513, 614, 740, 937 } },
    { "l'Oklahoma", new int[] { 405, 580, 918 } },
    { "l'Ontario", new int[] { 416, 519, 613, 647, 705, 807, 905 } },
    { "l'Orégon", new int[] { 503, 541, 971 } },
    { "la Pennsylvanie", new int[] { 215, 267, 412, 484, 570, 610, 717, 724,
      814, 878, 902 } },
    { "Puerto Rico", new int[] { 787 } },
    { "le Québec", new int[] { 418, 450, 514, 819 } },
    { "Rhode Island", new int[] { 401 } },
    { "le Saskatchewan", new int[] { 306 } },
    { "la Caroline du Sud", new int[] { 803, 843, 864 } },
    { "le Dakota du Sud", new int[] { 605 } },
    { "le Tennessee", new int[] { 423, 615, 865, 901, 931 } },
    { "le Texas", new int[] { 210, 214, 254, 281, 361, 409, 469, 512, 682,
      713, 806, 817, 830, 832, 903, 915, 940, 956, 972 } },
    { "les Iles Vierges US", new int[] { 340 } },
    { "l'Utah", new int[] { 435, 801 } },
    { "le Vermont", new int[] { 802 } },
    { "la Virginie", new int[] { 540, 571, 703, 757, 804 } },
    { "Washington", new int[] { 206, 253, 360, 425, 509, 564 } },
    { "la Virginie de l'Ouest", new int[] { 304 } },
    { "le Wyoming", new int[] { 307 } },
    { "le Territoire du Yukon", new int[] { 867 } },
  };
}
```

Cette servlet reçoit la requête WAP comme une requête GET HTTP normale. Dans sa méthode `doGet()`, elle fixe le type de contenu à `text/vnd.wap.wml`, récupère le paramètre code, localise la région correspondante *via* la table de recherche et termine en générant un document WML contenant la région. Les indicatifs régionaux viennent d'un tableau `data` saisi manuellement que l'on convertit en une table de `Properties` à l'initialisation de la servlet. Un exemple de sortie est montré à la *figure 6-4*.

C'est tout pour le WAP

Si vous voulez en savoir plus sur la création de sites destinés aux périphériques WAP, lisez *Learning WML & WMLScript* de Martin Frost (O'Reilly & Associates). Il existe aussi beaucoup de bons sites de ressources, dont *http://www.wap-resources.net*, *http://www.AnywhereYouGo.com* et *http://www.wirelessdevnet.com*. Il y a même la liste de diffusion *webjava-wireless* hébergée par Sun et ses archives à *http://archives.java.sun.com/archives/webjava-wireless.html*.

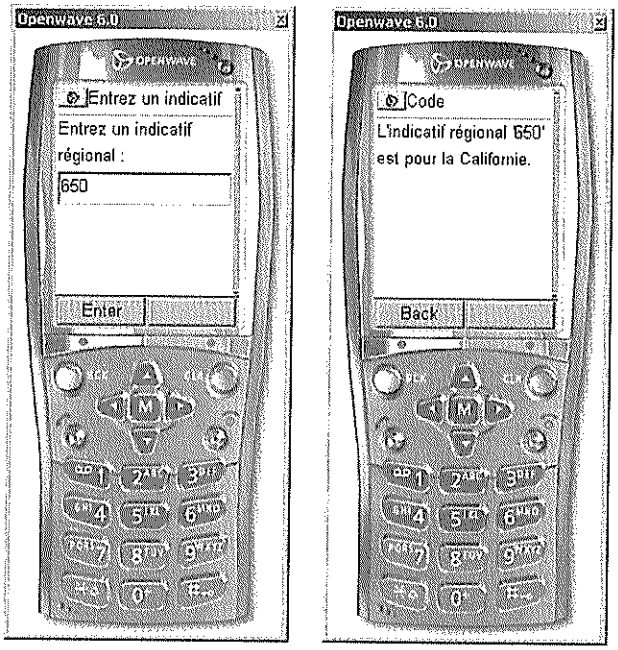


Figure 6-4. Utiliser WAP pour localiser les appelants

Images

Les gens aiment ce qui est visuel ; ils préfèrent voir, et pas seulement lire, les informations. Par conséquent, il est pratiquement impossible de trouver un site web qui n'utilise pas les images, et ceux qui ne les utilisent pas ne semblent pas très professionnels. Et de citer (traduit dans un langage de programmeur), « une image vaut bien mieux qu'un millier de mots ».

Heureusement, il est relativement simple pour une servlet d'envoyer une image en réponse. En fait, nous avons déjà vu une servlet qui fait exactement cela : la servlet `ViewResource` du chapitre 4, *Retrouver l'information*. Vous vous en souvenez peut-être, cette servlet peut retourner tout fichier qui se trouve sous la racine des documents du serveur. Lorsque ce fichier est une image, elle détecte ce fait avec la méthode `getMimeType()` et définit le type du contenu de sa réponse avec `setContentType()` avant d'envoyer les octets au client.

Cette technique impose que les fichiers image demandés aient déjà été sauvés sur le disque, ce qui n'est pas toujours le cas. Souvent, une servlet doit générer ou manipuler une image avant de l'envoyer aux clients. Imaginons, par exemple, une page web qui contienne l'image d'une horloge analogique affichant l'heure courante. Bien sûr, quelqu'un pourrait créer 720 images (60 minutes * 12 heures), les stocker sur le disque et utiliser une servlet qui affiche la bonne image. Mais ce quelqu'un ce n'est pas moi, et ce ne devrait pas être vous. À la place, le programmeur de servlet judicieux écrira une servlet qui génère dynamiquement l'image de l'horloge avec les aiguilles ; une servlet peut aussi charger l'image de l'horloge et y ajouter les aiguilles. Et, bien sûr, le programmeur

futé met la servlet en cache (pour une minute) afin d'économiser des cycles processeur du serveur.

Il y a beaucoup d'autres raisons pour qu'une servlet retourne une image. En générant des images, la servlet peut afficher des choses telles qu'un graphique financier à la minute, le score courant d'un match de football (y compris les icones représentant les joueurs sur le terrain) ou une représentation graphique des jus de fruits disponibles dans le distributeur. En manipulant des images préexistantes, une servlet peut même faire encore plus. Elle peut dessiner par-dessus, changer leur couleur, leur taille, leur apparence ou combiner plusieurs images en une seule.

Générer des images

Supposons que vous ayez une image sous forme de pixels et que vouliez l'envoyer à quelqu'un. Comment faites-vous cela ? Supposons que ce soit une image en vraies couleurs (24 bits, 3 octets par pixels) et qu'elle fasse 100 x 100 pixels. Vous pouvez prendre une approche évidente qui consiste à envoyer un pixel à la fois, dans un flot de 30 000 octets. Mais est-ce suffisant ? Comment le receveur sait-il ce qu'il doit faire des 30 000 octets ? La réponse est qu'il ne le sait pas. Vous devez aussi indiquer que vous envoyez des valeurs de pixel en vraies couleurs, que vous commencez par le coin en haut à gauche, que vous envoyez ligne par ligne et que chaque ligne fait 100 pixels de large. Et que se passe-t-il si vous décidez d'envoyer moins d'octets en utilisant la compression ? Vous devez indiquer le type de compression que vous utilisez de façon à ce que le receveur puisse décomprimer l'image. Cela devient tout d'un coup un problème bien plus complexe.

Heureusement, ce problème a été résolu, et cela de plusieurs manières. Chaque format (GIF, JPEG, PNG, TIFF, etc.) représente une solution. Chaque format définit une manière standard de coder une image de façon à ce qu'elle puisse être ensuite décodée pour être affichée ou manipulée. Chaque technique de codage a ses avantages et ses limitations. Par exemple, la compression utilisée dans GIF est excellente pour les images générée par un ordinateur, mais ce format est limité à 256 couleurs. La compression utilisée dans JPEG fonctionne très bien sur des images photo-réalistes qui contiennent des millions de couleurs, mais uniquement parce qu'elle utilise une compression « avec perte » qui estompe les détails de la photo. PNG est un format d'encodage relativement récent destiné à remplacer GIF car il est plus petit, supporte plusieurs millions de couleurs, emploie une compression « sans perte » et dispose d'un canal alpha pour les effets de transparence ; en plus il est libre de tout problème de copyright contrairement à GIF.[3]

Comprendre le codage d'une image aide à comprendre comment les servlets les traitent. Une servlet comme `ViewResource` peut retourner une image préexistante en envoyant sa représentation codée non modifiée au client ; le navigateur décode l'image pour son affichage. Mais une servlet qui génère ou modifie une image doit construire une représentation interne de cette image, la manipuler et ensuite la coder avant de l'envoyer au client.

3. Pour plus d'informations sur PNG, voir *http://graphicswiz.com/png*.

Un « Bonjour tout le monde » graphique

L'*exemple 6-6* montre un exemple simple d'une servlet qui génère et retourne une image GIF. Cette image affiche « Bonjour tout le monde ! », comme montré à la *figure 6-5*.

Exemple 6-6. Bonjour tout le monde, version graphique

```
import java.io.*;
import java.awt.*;
import javax.servlet.*;
import javax.servlet.http.*;

import Acme.JPM.Encoders.GifEncoder;

public class HelloWorldGraphics extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();  // Sortie binaire !

    Frame frame = null;
    Graphics g = null;

    try {
      // Créer un Frame caché.
      frame = new Frame();
      frame.addNotify();

      // Obtenir une région graphique à l'aide du Frame.
      Image image = frame.createImage(620, 60);
      g = image.getGraphics();

      // Dessiner "Bonjour tout le monde !" dans le contexte graphique
      // hors écran.
      g.setFont(new Font("Serif", Font.ITALIC, 48));
      g.drawString("Bonjour tout le monde !", 10, 50);

      // Convertir l'image hors écran en GIF et l'envoyer au client.
      res.setContentType("image/gif");
      GifEncoder encoder = new GifEncoder(image, out);
      encoder.encode();
    }
    finally {
      // Libérer les ressources.
      if (g != null) g.dispose();
      if (frame != null) frame.removeNotify();
    }
  }
}
```

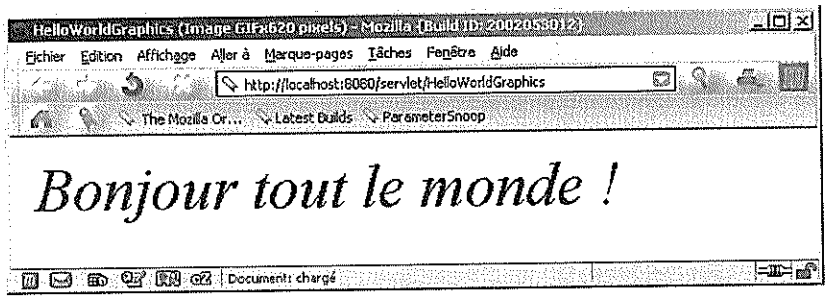


Figure 6-5. Bonjour tout le monde, version graphique

Bien que cette servlet utilise le paquetage `java.awt`, elle n'affiche jamais une fenêtre sur le serveur. Elle n'affiche pas non plus une fenêtre sur le client. Elle effectue tout son travail dans un contexte graphique hors écran et laisse le navigateur afficher l'image. La stratégie est la suivante : créer une image hors écran, obtenir son contexte graphique, dessiner l'image et ensuite coder le résultat pour l'envoi au client.

La création d'une image hors écran se fait en plusieurs étapes. En Java, une image est représentée par la classe `java.awt.Image`. Malheureusement, dans le JDK 1.1, un objet `Image` ne peut pas être instancié directement *via* un constructeur. Il doit être obtenu par une méthode comme `createImage()` fournie par `Component` ou `getImage()` de `Toolkit`. Puisque nous créons une nouvelle image, nous utilisons `createImage()`. Notez qu'avant qu'un composant puisse créer une image, son pair natif doit déjà exister. Ainsi, pour créer notre `Image`, nous devons créer un `Frame`, créer son pair par un appel à `addNotify()` et ensuite l'utiliser pour créer notre `Image`[4].

Dans le JDK 1.2, le processus est simplifié et il est possible de créer un `Image` directement en construisant un nouveau `java.awt.image.BufferedImage`. Cependant, les exemples de ce chapitre utilisent la technique `frame.createImage()` afin de conserver une portabilité maximale sur les version du JDK.

Une fois l'image obtenue, nous dessinons dessus en utilisant son contexte graphique, qui peut être retrouvé par un appel à la méthode `getGraphics()` de `Image`. Dans cet exemple, nous dessinons simplement une chaîne de caractères.

Après avoir dessiné dans le contexte graphique, nous appelons `setContentType()` pour définir le type MIME à `image/gif` puisque nous allons utiliser le codage GIF. Pour les exemples de ce livre, nous utilisons un codeur GIF écrit par Jef Poskanzer. Il est bien écrit et disponible gratuitement avec les sources à *http://www.acme.com*.

Notez que l'algorithme de compression LZW utilisé dans le codage GIF est protégé par les brevets d'Unisys et d'IBM qui, d'après la Free Software Foundation, interdit la créa-

4. Pour certains serveurs web tournant sur Unix, le pair natif du Frame doit être créé dans un serveur X. Ainsi, pour des performances optimales, assurez-vous que la variable d'environnement `DISPLAY` (qui spécifie le serveur X à utiliser) n'est pas définie ou qu'elle indique un serveur X local. Assurez-vous aussi que le serveur web a les droits d'accès au serveur X, ce qui peut demander l'utilisation de xhost ou xauth. Les machines serveur sans moniteur et sans serveur X peuvent utiliser Xvfb (*X virtual frame buffer*) pour gérer le graphisme ; assurez-vous simplement que `DISPLAY` désigne le serveur Xvfb.

tion de logiciels gratuits acceptant le format GIF. Le codeur Acme GIF utilise la compression LZ, il évite donc peut-être le brevet. Pour plus d'informations, voir *http://www.fsf.org/philosophy/gif.html* et *http://www.burnallgifs.org*. Bien entendu, une servlet peut coder son `Image` dans tout format d'image. Pour le contenu web, JPEG et PNG sont les alternatives à GIF les plus viables. Il existe plusieurs codeurs JPEG et PNG. Pour les utilisateurs du JDK 1.2 et ultérieurs, le paquetage `com.sun.image.codec.jpeg` contient un codeur JPEG ; les codeurs pour PNG et d'autres formats sont disponibles pour le JDK 1.2 dans l'API Java Advanced Imaging à *http://java.sun.com/products/java-media/jai*. Pour les utilisateurs ayant aussi besoin d'un support pour le JDK 1.1, il existe l'outil JIMI (*Java Image Management Interface*), précédemment un produit commercial d'Activated Intelligence et maintenant une bibliothèque Sun disponible gratuitement à *http://java.sun.com/products/jimi*.

Pour coder l'image, nous créons un objet `GifEncoder`, en lui passant l'objet `Image` et le `ServletOutputStream` pour la servlet. Lorsque nous appelons `encode()` sur l'objet `GifEncoder`, l'image est codée et envoyée au client.

Après avoir envoyé l'image, la servlet fait ce que toute servlet bien faite doit faire : elle libère ses ressources graphiques. Elles seraient récupérées automatiquement pendant le « ramassage des miettes », mais les libérer immédiatement aide les systèmes ayant des ressources limitées. Le code pour libérer les ressources est placé dans un bloc `finally` afin de garantir son exécution, même lorsque la servlet lance une exception.

Un graphique généré dynamiquement

Examinons maintenant une servlet qui génère une image plus intéressante. L'*exemple 6-7* crée un graphique en barres qui compare les pommes et les oranges (qui a dit que ce n'est pas possible ?), en fonction de leur consommation annuelle.

Exemple 6-7. Graphique de comparaison des pommes et des oranges

```
import java.awt.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import Acme.JPM.Encoders.GifEncoder;

import javachart.chart.*;  // De Visual Engineering.

public class SimpleChart extends HttpServlet {

  static final int WIDTH = 450;
  static final int HEIGHT = 320;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException ,IOException {
    ServletOutputStream out = res.getOutputStream();

    Frame frame = null;
    Graphics g = null;
```

Exemple 6-7. Graphique de comparaison des pommes et des oranges (suite)

```
try {
  // Créer un graphique simple.
  BarChart chart = new BarChart("Pommes et oranges");

  // Lui donner un titre.
  chart.getBackground().setTitleFont(new Font("Serif", Font.PLAIN, 24));
  chart.getBackground().setTitleString(
      "Comparer les pommes et les oranges");

  // Afficher, placer et personnaliser la légende.
  chart.setLegendVisible(true);
  chart.getLegend().setL1X(0.4);  // Normalisé à partir du bas gauche.
  chart.getLegend().setL1Y(0.75); // Normalisé à partir du bas gauche.
  chart.getLegend().setIconHeight(0.04);
  chart.getLegend().setIconWidth(0.04);
  chart.getLegend().setIconGap(0.02);
  chart.getLegend().setVerticalLayout(false);

  // Lui donner des données et des étiquettes.
  double[] appleData = {950, 1005, 1210, 1165, 1255};
  chart.addDataset("Pommes", appleData);

  double[] orangeData = {1435, 1650, 1555, 1440, 1595};
  chart.addDataset("Oranges", orangeData);

  String[] labels = {"1993", "1994", "1995", "1996", "1997"};
  chart.getXAxis().addLabels(labels);

  // Les pommmes en rouge et les oranges en orange.
  chart.getDatasets()[0].getGc().setFillColor(Color.red);
  chart.getDatasets()[1].getGc().setFillColor(Color.orange);

  // Nommer les axes.
  chart.getXAxis().setTitleString("Années");
  chart.getYAxis().setTitleString("Tonnes consommées");

  // Redimensionner.
  chart.resize(WIDTH, HEIGHT);

  // Créer un cadre caché.
  frame = new Frame();
  frame.addNotify();

  // Obtenir une région graphique de la bonne taille à l'aide du Frame.
  Image image = frame.createImage(WIDTH, HEIGHT);
  g = image.getGraphics();
```

Exemple 6-7. Graphique de comparaison des pommes et des oranges (suite)

```
        // Demander au graphique de se dessiner lui-même dans le contexte
        // graphique hors écran.
        chart.drawGraph(g);

        // Coder et retourner ce qui a été peint.
        res.setContentType("image/gif");
        GifEncoder encoder = new GifEncoder(image, out);
        encoder.encode();
      }
      finally {
        // Libérer les ressources.
        if (g != null) g.dispose();
        if (frame != null) frame.removeNotify();
      }
    }
}
```

La *figure 6-6* montre les résultats. Il n'est pas vraiment nécessaire que ce graphique soit généré dynamiquement, mais il va nous permettre d'arriver au but sans trop de code. Si vous le voulez, imaginez que la servlet dessine toutes les minutes un graphique des valeurs boursières ou de la charge récente du serveur.

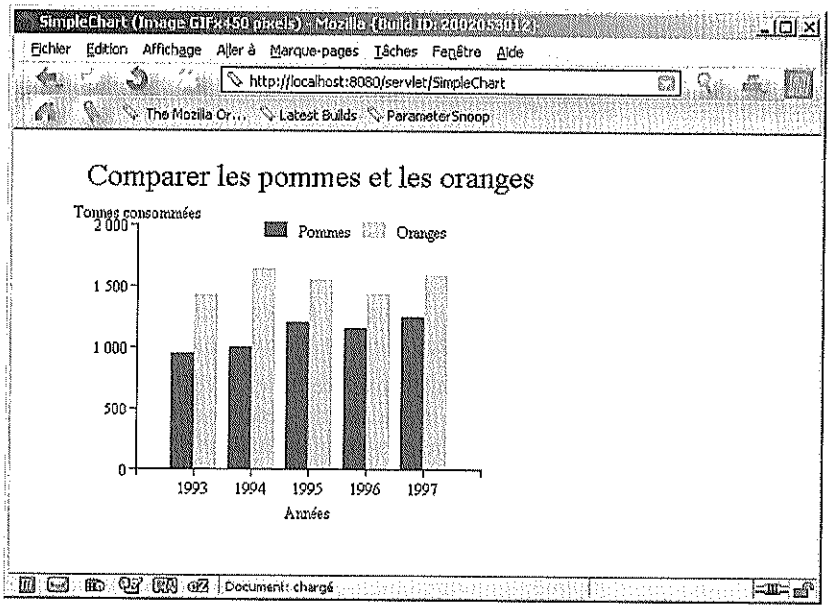


Figure 6-6. Graphique comparant les pommes et les oranges

Les bases sont les mêmes : créer une image hors écran et obtenir son contexte graphique, dessiner et coder l'image pour la transmettre au client La différence est que la servlet construit un objet `BarChart` pour effectuer le dessin. Il existe plus d'une dizaine de

paquetages de réalisation de graphiques en Java. La classe BarChart de cet exemple vient du paquetage KavaChart (précédemment JavaChart) de Visual Engineering, disponible à *http://www.ve.com/javachart*. C'est un produit commercial, mais pour les lecteurs de ce livre, ils ont accepté que nous utilisions gratuitement une partie de l'API présentée ci-dessus. Le paquetage KavaChart inclut aussi un ensemble gratuit d'applets de génération de graphiques disponible à *http://www.ve.com/kavachart/servlets.html*. JClass Chart de Sitraka à *http://www.sitraka.com* (précédemment KL Group) est un autre bon paquetage graphique.

Composer des images

Jusqu'à présent, nous avons dessiné notre graphique sur des images vides. Dans cette section, nous expliquons comment prendre des images existantes et dessiner par-dessus ou les combiner à d'autres images. Nous examinons aussi la gestion des erreurs dans les servlets qui retournent des images.

Dessiner par-dessus une image

Il est parfois utile que la servlet dessine par-dessus une image existante. Un bon exemple serait une servlet de localisation qui sait où se trouve chaque employé. Lors de la recherche d'un employé, elle pourrait dessiner un point rouge sur son bureau.

Une technique évidente mais décevante pour dessiner par-dessus une image existante est de retrouver l'Image avec `Toolkit.getDefaultToolkit().getImage(imagename)`, d'obtenir son contexte graphique par un appel à la méthode `getGraphics()` de Image, et d'utiliser ensuite le contexte graphique retourné pour dessiner par-dessus l'image. Malheureusement, ce n'est pas si facile. En effet, vous ne pouvez pas utiliser `getGraphics()` à moins que l'image n'ait été créée avec la méthode `createImage()` de Component. Avec l'AWT, vous devez toujours avoir un pair natif en coulisses qui effectue le rendu.

Voici ce que vous devez faire : retrouver une image existante *via* la méthode `Toolkit.getDefaultToolkit().getImage(imagename)` et ensuite lui indiquer de se dessiner elle-même dans un autre contexte graphique créé avec la méthode `createImage()` de Component, comme montré dans les deux exemples précédents. Maintenant, vous pouvez utiliser ce contexte graphique pour dessiner par-dessus l'image originale.

L'*exemple 6-8* clarifie cette technique avec un exemple. C'est une servlet qui écrit « CONFIDENTIEL » par-dessus chaque image qu'elle retourne. Le nom de l'image est passé à la servlet dans l'information de chemin supplémentaire. ; la servlet charge l'image à partir de l'emplacement correspondant sous la racine des documents du serveur à l'aide de la méthode `getResource()` afin de supporter une exécution distribuée.

Exemple 6-8. Dessiner par dessus une image pour la marquer confidentielle

```
import java.awt.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Exemple 6-8. Dessiner par dessus une image pour la marquer confidentielle (suite)

```
import com.oreilly.servlet.ServletUtils;

import Acme.JPM.Encoders.GifEncoder;

public class Confidentializer extends HttpServlet {

  Frame frame = null;

  public void init() throws ServletException {
    // Construire un Frame caché réutilisable.
    frame = new Frame();
    frame.addNotify();
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();
    Graphics g = null;

    try {
      // Prendre l'emplacement de l'image dans l'information de chemin
      // supplémentaire. Utiliser ServletUtils (chapitre 4) pour plus
      // de sûreté.
      URL source =
        ServletUtils.getResource(getServletContext(), req.getPathInfo());

      // Charger l'image (convertir des octets en un objet Image).
      MediaTracker mt = new MediaTracker(frame); // frame agit comme un
                                                 // ImageObserver.
      Image image = Toolkit.getDefaultToolkit().getImage(source);
      mt.addImage(image, 0);
      try {
        mt.waitForAll();
      }
      catch (InterruptedException e) {
        res.sendError(res.SC_INTERNAL_SERVER_ERROR,
                "Interruption du chargement de l'image : " +
                ServletUtils.getStackTraceAsString(e));
        return;
      }

      // Obtenir la largeur et la hauteur.
      int w = image.getWidth(frame);
      int h = image.getHeight(frame);

      // S'assurer que les données de l'image sont valides.
      if (w <= 0 || h <= 0) {
        res.sendError(res.SC_NOT_FOUND, "L'information de chemin " +
                      "supplémentaire doit désigner une image valide ");
```

Exemple 6-8. Dessiner par dessus une image pour la marquer confidentielle (suite)

```
        return;
      }

      // Construire un contexte graphique hors écran de taille équivalente.
      Image offscreen = frame.createImage(w, h);
      g = offscreen.getGraphics();

      // Dessiner l'image dans le contexte graphique hors écran.
      g.drawImage(image, 0, 0, frame);

      // Écrire CONFIDENTIEL en haut.
      g.setFont(new Font("Monospaced", Font.BOLD | Font.ITALIC, 30));
      g.drawString("CONFIDENTIEL", 10, 30);

      // Coder l'image hors écran en GIF et l'envoyer au client.
      res.setContentType("image/gif");
      GifEncoder encoder = new GifEncoder(offscreen, out);
      encoder.encode();
    }
    finally {
      // Libérer les ressources.
      if (g != null) g.dispose();
    }
  }

  public void destroy() {
    // Libérer les ressources.
    if (frame != null) frame.removeNotify();
  }
}
```

Un exemple de sortie est montré à la *figure 6-7*.

Vous pouvez voir que cette servlet effectue chaque étape exactement comme nous l'avons décrit, avec quelques autres tâches. La servlet crée son `Frame` invisible dans la méthode `init()`. Créer le `Frame` une fois et le réutiliser est une optimisation que l'on n'avait pas réalisée précédemment pour des raisons de simplicité. Pour chaque requête, la servlet commence par retrouver le nom de l'image existante à partir de l'information de chemin supplémentaire et convertit le chemin en une ressource à l'aide de `getResource()`. Ensuite, elle retrouve une référence à l'image avec la méthode `getImage()` de `Toolkit` et elle la charge physiquement en mémoire avec un `MediaTracker`. Normalement, une image peut se charger de façon asynchrone pendant qu'elle s'affiche partiellement sans aucun problème. Mais dans le cas présent, nous affichons l'image une seule fois et devons garantir qu'elle est totalement chargée. Ensuite, la servlet obtient la largeur et la hauteur de l'image chargée et crée l'image hors écran correspondante. Enfin, le moment tant attendu : l'image chargée est dessinée par-dessus la nouvelle image vide. Après cela, la servlet écrit son grand « CONFIDENTIEL » et code l'image pour l'envoi.

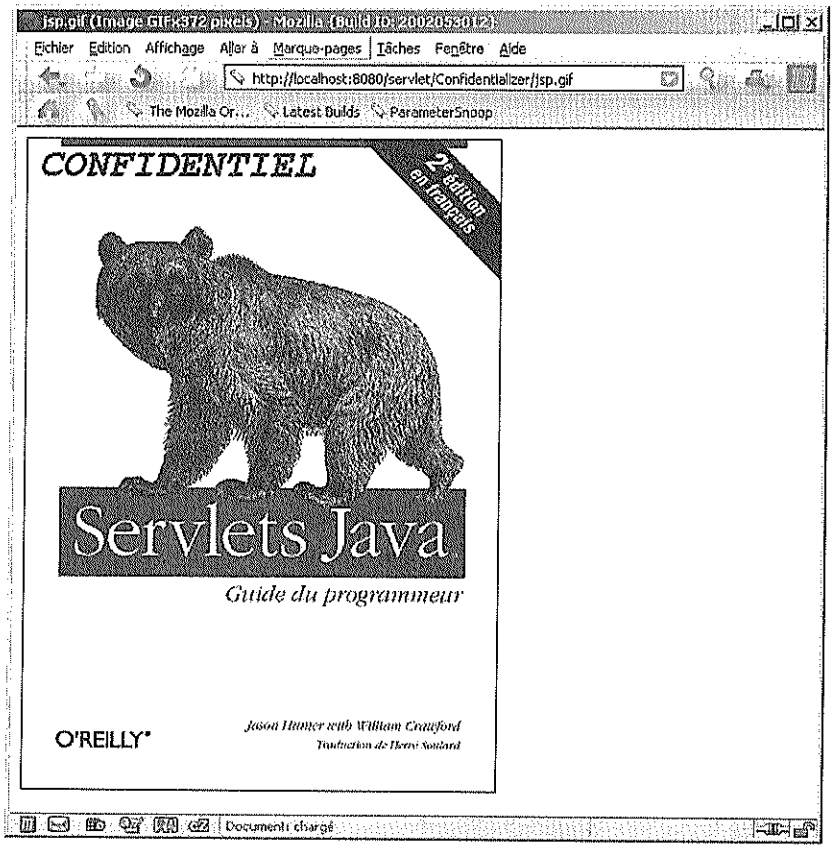


Figure 6-7. Dessiner par dessus une image pour la marquer confidentielle

Notez comment cette servlet gère les conditions d'erreur en appelant `sendError()`. Lors du renvoi d'images, il est difficile de faire mieux. Cette approche permet au serveur de faire ce qui lui semble opportun.

Combiner des images

Une servlet peut aussi combiner des images en une seule. En utilisant cette possibilité, une servlet de localisation pourrait afficher la tête souriante d'un employé au-dessus de son bureau, au lieu d'un point rouge. La technique utilisée pour combiner des images est similaire à celle utilisée pour dessiner par-dessus une image : les images appropriées sont chargées, elles sont dessinées par-dessus un objet `Image` correctement crée et cette image est codée pour l'envoi.

L'*exemple 6-9* montre comment réaliser une servlet qui affiche un compteur de hits sous forme d'une séquence d'images individuelles de chiffre combinées en une seule grande image. Les images de chiffre qu'elle utilise sont disponibles à *http://member.apcug.org/counter/*, ainsi que beaucoup d'autres de style différent.

Exemple 6-9. Combiner des images pour réaliser un compteur graphique

```
import java.awt.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

import Acme.JPM.Encoders.GifEncoder;

public class GraphicalCounter extends HttpServlet {

  public static final String DIR = "/images/odometer";
  public static final String COUNT = "314159";

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();

    Frame frame = null;
    Graphics g = null;

    try {
      // Obtenir le compteur à afficher, doit être la seule valeur dans la
      // chaîne d'interrogation. Ou utiliser la valeur par défaut.
      String count = (String)req.getQueryString();
      if (count == null) count = COUNT;

      int countlen = count.length();
      Image images[] = new Image[countlen];

      for (int i = 0; i < countlen; i++) {
        URL imageSrc =
         getServletContext().getResource(DIR + "/" + count.charAt(i) + ".GIF");
        if (imageSrc == null) {
          imageSrc = new URL("file:");  // Emplacement, gérer les
        }                               // erreurs ensuite.
        images[i] = Toolkit.getDefaultToolkit().getImage(imageSrc);
      }

      // Créer un Frame caché.
      frame = new Frame();
      frame.addNotify();

      // Charger les images.
      MediaTracker mt = new MediaTracker(frame);
      for (int i = 0; i < countlen; i++) {
        mt.addImage(images[i], i);
      }
```

Exemple 6-9. Combiner des images pour réaliser un compteur graphique (suite)

```
    try {
      mt.waitForAll();
    }
    catch (InterruptedException e) {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
              "Interruption du chargement des images : " +
              ServletUtils.getStackTraceAsString(e));
      return;
    }

    // Contrôler les problèmes de chargement des images.
    if (mt.isErrorAny()) {
      // Nous avons eu un problème, trouver quelle(s) image(s).
      StringBuffer problemChars = new StringBuffer();
      for (int i = 0; i < countlen; i++) {
        if (mt.isErrorID(i)) {
          problemChars.append(count.charAt(i));
        }
      }
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
              "Impossible de charger une image pour ces caractères : " +
              problemChars.toString());
      return;
    }

    // Obtenir la taille cumulée des images.
    int width = 0;
    int height = 0;
    for (int i = 0; i < countlen; i++) {
      width += images[i].getWidth(frame);
      height = Math.max(height, images[i].getHeight(frame));
    }

    // Obtenir une région graphique, en utilisant le Frame.
    Image image = frame.createImage(width, height);
    g = image.getGraphics();

    // Dessiner les images.
    int xindex = 0;
    for (int i = 0; i < countlen; i++) {
      g.drawImage(images[i], xindex, 0, frame);
      xindex += images[i].getWidth(frame);
    }

    // Coder et retourner le tout.
    res.setContentType("image/gif");
    GifEncoder encoder = new GifEncoder(image, out);
    encoder.encode();
  }
```

Exemple 6-9. Combiner des images pour réaliser un compteur graphique (suite)

```
    finally {
      // Libérer les ressources
      if (g != null) g.dispose();
      if (frame != null) frame.removeNotify();
    }
  }
}
```

La sortie est visible à la *figure 6-8*.

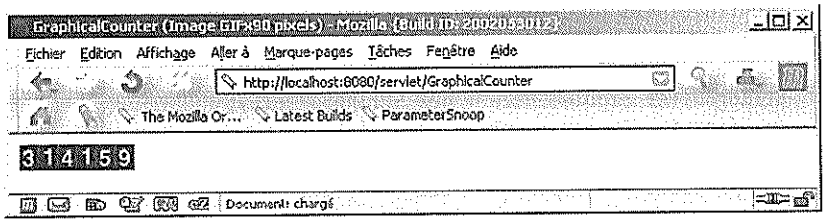


Figure 6-8. Combiner des images pour créer un compteur graphique

Cette servlet reçoit le nombre à afficher en lisant sa chaîne d'interrogation. Pour chaque chiffre du compteur, elle retrouve et charge l'image correspondante à partir du répertoire donné par DIR. (DIR est toujours sous la racine des documents du serveur. Il est donné sous forme de chemin virtuel et converti en un chemin de ressources abstrait.) Ensuite, elle calcule la largeur et la hauteur combinées de toutes ces images et construit l'image hors écran correspondante. La servlet dessine chaque image de chiffre dans cette image hors écran de gauche à droite. Enfin, elle code l'image pour l'envoi.

Pour qu'elle soit utilisable, cette servlet doit être appelée par une autre servlet qui connaît le nombre de hits à afficher et le place dans la chaîne d'interrogation. Par exemple, elle pourrait être appelée par une page JSP ou toute autre page créée dynamiquement à l'aide d'une syntaxe similaire à la suivante :

```
<IMG SRC="/servlet/GraphicalCounter?121672">
```

Cette servlet gère les conditions d'erreur de la même façon que la précédente, en appelant `sendError()` et en laissant le serveur se comporter de manière adéquate.

Effets sur une image

Nous avons vu comment les servlets peuvent créer et combiner des images. Dans cette section, nous examinons comment les servlets peuvent aussi appliquer des effets spéciaux sur des images. Par exemple, une servlet peut diminuer le temps de transmission d'une image en réduisant sa taille avant l'envoi. Ou, elle peut ajouter un ombrage spécial afin que l'image ressemble à un bouton. Nous prenons pour exemple une servlet qui convertit en niveaux de gris une image en couleurs.

Convertir une image en niveaux de gris

L'*exemple 6-10* montre une servlet qui convertit une image en niveaux de gris avant de la retourner. La servlet effectue cet effet sans jamais réellement créer un contexte graphique hors écran. À la place, elle crée l'image en utilisant un `ImageFilter` spécial. (Nous ne montrerons pas les images avant et après, elles ne seraient pas très convaincantes dans un livre en noir et blanc.)

Exemple 6-10. Conversion d'une image en niveaux de gris

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

import Acme.JPM.Encoders.*;

public class DeColorize extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("image/gif");
    ServletOutputStream out = res.getOutputStream();

    // Obtenir l'emplacement de l'image à partir de l'information de chemin.
    URL source = ServletUtils.getResource(getServletContext(),
                                          req.getPathInfo());
    if (source == null) {
      res.sendError(res.SC_NOT_FOUND,
          "L'information de chemin supplémentaire doit désigner une image");
      return;
    }

    // Construire un Frame non affiché.
    // Ne pas appeler addNotify() car le pair n'est pas nécessaire.
    Frame frame = new Frame();

    // Charger l'image.
    Image image = Toolkit.getDefaultToolkit().getImage(source);
    MediaTracker mt = new MediaTracker(frame);
    mt.addImage(image, 0);
    try {
      mt.waitForAll();
    }
    catch (InterruptedException e) {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
```

Exemple 6-10. Conversion d'une image en niveaux de gris (suite)

```
            "Interruption pendant le chargement de l'image : " +
            ServletUtils.getStackTraceAsString(e));
      return;
    }

    // Obtenir la taille de l'image.
    int width = image.getWidth(frame);
    int height = image.getHeight(frame);

    // S'assurer que les données de l'image sont valides.
    if (width <= 0 || height <= 0) {
      res.sendError(res.SC_NOT_FOUND, "L'information de chemin " +
                    "supplémentaire doit désigner une image valide ");
      return;
    }

    // Créer une image correspondante, la passer par un filtre.
    Image filtered = frame.createImage(
      new FilteredImageSource(image.getSource(),
                              new GrayscaleImageFilter()));

    // Coder et retourner l'image filtrée.
    GifEncoder encoder = new GifEncoder(filtered, out);
    encoder.encode();
  }
}
```

La majorité du code de cette servlet correspond à l'exemple `Confidentializer`. Voici la différence principale :

```
// Créer une image correspondante, la passer par un filtre.
Image filtered = frame.createImage(
  new FilteredImageSource(image.getSource(),
                          new GrayscaleImageFilter()));
```

Cette servlet n'utilise pas la méthode `createImage(int, int)` de `Component` utilisée jusque-là. À la place, elle tire parti de la méthode `createImage(ImageProducer)` de `Component`. La servlet crée un producteur d'image avec un `FilteredImageSource` qui passe l'image au travers d'un `GrayscaleImageFilter`. Ce filtre convertit chaque pixel en couleur en son équivalent en niveau de gris. L'image est donc convertie en niveaux de gris au fur et à mesure qu'elle est créée. Le code de `GrayscaleImageFilter` est donné à l'*exemple 6-11*.

Exemple 6-11. La classe GrayscaleImageFilter

```
import java.awt.*;
import java.awt.image.*;

public class GrayscaleImageFilter extends RGBImageFilter {
```

Exemple 6-11. La classe GrayscaleImageFilter (suite)

```
  public GrayscaleImageFilter() {
    canFilterIndexColorModel = true;
  }

  // Convertir les pixels de couleur en niveaux de gris.
  // L'algorithme correspond à la spécification NTSC.
  public int filterRGB(int x, int y, int pixel) {

    // Obtenir l'intensité RVB moyenne.
    int red = (pixel & 0x00ff0000) >> 16;
    int green = (pixel & 0x0000ff00) >> 8;
    int blue = pixel & 0x000000ff;

    int luma = (int) (0.299 * red + 0.587 * green + 0.114 * blue);

    // Retourner la valeur luma comme la valeur de chaque composante RVB.
    // Note: Alpha (transparence) est toujours au max (non transparente).
    return (0xff << 24) | (luma << 16) | (luma << 8) | luma;
  }
}
```

Pour chaque valeur de la table de couleurs, ce filtre reçoit une valeur de pixel et retourne une nouvelle valeur de pixel filtrée. En mettant la variable `canFilterIndexColorModel` à `true`, nous indiquons que ce filtre opère sur la table de couleurs et non sur des valeurs de pixel. Une valeur de pixel est un entier sur 32 bits, le premier octet représente la valeur alpha (transparence), le second octet l'intensité du rouge, le troisième celle du vert et le quatrième celle du bleu. Pour convertir une valeur de pixel en niveaux de gris, les intensités de rouge, de vert et de bleu doivent être mises à des valeurs identiques. Nous pouvons faire une moyenne des valeurs de rouge, de vert et de bleu et l'utiliser pour chaque intensité de couleur. Cela convertit l'image en niveaux de gris. Cependant, pour prendre en compte la façon dont les gens perçoivent réellement les couleurs (et d'autres facteurs), il faut une moyenne pondérée. Les poids 0,299, 0,587 et 0,114 utilisés ici correspondent à celles utilisées par le National Television Systems Committee pour la télévision en noir et blanc. Pour plus d'informations, lisez le livre de Charles A. Poynton *A Technical Introduction to Digital Video* (Wiley) et allez voir le site web *http://www.color.org*.

Mettre en cache une image convertie

Le processus de création et de codage d'une image peut coûter cher, prenant du temps et des cycles CPU du serveur. Le fait de mettre en cache les images améliore souvent les performances. Au lieu d'effectuer le travail à chaque requête, les résultats peuvent être sauvés et envoyés à nouveau lors des requêtes suivantes. L'idée de l'horloge mentionnée précédemment en est un parfait exemple. L'image de l'horloge doit être créée au plus une fois par minute. Toute requête pendant cette minute peut renvoyer la même image. Un graphique de comptabilisation pour un vote en est un autre exemple. Il peut être créé une fois et modifié uniquement lorsque de nouveaux votes arrivent.

La classe mère `com.oreilly.servlet.CacheHttpServlet` du chapitre 3, *Cycle de vie d'une servlet*, apporte un mécanisme de cache simple pour les images et pour le texte. Une servlet générant l'image de l'horloge peut étendre `CacheHttpServlet` et implémenter une méthode `getLastModified()` qui renvoie l'heure courante arrondie à la minute inférieure. Une servlet générant le graphique pour les votes peut implémenter une méthode `getLastModified()` qui retourne l'heure d'arrivée du dernier vote.

Pour notre exemple, notre servlet `DeColorize` peut étendre `CacheHttpServlet` et implémenter une méthode `getLastModified()` qui retourne la date de dernière modification de l'image. Malheureusement, `CacheHttpServlet` ne met en cache que la dernière réponse, apportant peu de bénéfices dans le cas très probable où `DeColorize` est appelée sur plus d'une image. `DeColorize` doit donc probablement employer un algorithme de cache plus complexe prenant en charge les images multiples. Le cycle de vie d'une servlet rend cela plutôt simple. Notre nouvelle servlet `DeColorize` sauve chaque image convertie sous forme de tableau d'octets dans une table de hachage indexée par le nom de l'image. Tout d'abord, notre servlet doit créer une variable d'instance de `Hashtable`. Elle doit être déclarée en-dehors de `doGet()` :

```
Hashtable gifs = new Hashtable();
```

Pour remplir cette table de hachage, nous devons capturer le graphisme codé. Donc, au lieu de donner le `ServletOutputStream` à `GifEncoder`, nous lui donnons un `ByteArrayOutputStream`. Ensuite, lorsque nous codons l'image avec `encode()`, l'image codée est stockée dans le `ByteArrayOutputStream`. Enfin, nous stockons les octets capturés dans la table de hachage et les écrivons ensuite sur le `ServletOutputStream` pour envoyer l'image au client. Voici le nouveau code pour coder, stocker et retourner l'image filtrée :

```
// Coder, stocker et retourner l'image filtrée.
ByteArrayOutputStream baos = new ByteArrayOutputStream(1024);// 1K au début
GifEncoder encoder = new GifEncoder(filtered, baos);
encoder.encode();
gifs.put(source, baos);
baos.writeTo(out);
```

Cela remplit la table de hachage avec les images codées indexées par leur nom. Maintenant, nous pouvons aller directement dans le cache lorsque l'on demande à la servlet de retourner une image précédemment codée. Le code suivant doit être placé immédiatement après le code exécuté si `source==null` :

```
// Raccourci si cela a déjà été fait.
if (gifs.containsKey(source)) {
  ByteArrayOutputStream baos = (ByteArrayOutputStream) gifs.get(source);
  baos.writeTo(out);
  return;
}
```

Avec ces modifications, toute image trouvée dans le cache est retournée rapidement, directement à partir de la mémoire.

Bien sûr mettre en cache de multiples images a tendance à consommer de grandes quantités de mémoire. Mettre en cache une seule image posera rarement des problèmes, mais une servlet comme celle-ci devrait utiliser des mécanismes de « nettoyage ». Par exemple, elle pourrait ne mettre en cache que les 100 dernières images demandées. Une version plus robuste de la servlet pourrait examiner les dates du fichier afin de s'assurer

que l'image originale n'a pas été modifiée depuis la dernière mise en cache de la version en noir et blanc.

Contenu compressé

Le paquetage `java.util.zip` est arrivé avec le JDK 1.1. Il contient des classes qui supportent la lecture et l'écriture des formats de compression GZIP et ZIP. Bien que ces classes aient été ajoutées pour supporter les fichiers JAR (*Java Archives*), elles fournissent aussi une façon pratique et standard pour que les servlets envoient du contenu compressé.

L'utilisateur final ne voit pas de différences quand il reçoit un contenu compressé car ce dernier a été décompressé par le navigateur avant son affichage. Bien qu'il n'y ait pas de différence, cela peut améliorer l'expérience de l'utilisateur final car le temps de téléchargement du contenu est diminué. Pour du contenu hautement comprimable comme du HTML, la compression peut diminuer le temps de transmission d'un ordre de grandeur. Quelle astuce ! Gardez quand même à l'esprit que la compression dynamique du contenu oblige le serveur à effectuer un travail plus important. Toute amélioration du temps de transmission doit donc être comparée à la réduction des performances du serveur.

Vous devriez être maintenant familier avec l'idée qu'une servlet peut envoyer un en-tête `Content-Type` dans sa réponse afin d'indiquer au client le type de l'information retournée. Pour envoyer du contenu compressé, une servlet doit aussi envoyer un en-tête `Content-Encoding` indiquant au client le schéma de codage du contenu. Dans la spécification HTTP 1.0, les possibilités de schéma de codage sont `gzip` (ou `x-gzip`), `compress` (ou `x-compress`) et `deflate`.

Tous les clients ne comprennent pas ces codages. Pour indiquer au serveur quels sont les schémas compris, un client envoie un en-tête `Accept-Encoding` qui spécifie les schémas de codage acceptables dans une liste séparés par des virgules. La plupart des navigateurs ne fournissent pas encore cet en-tête, même ceux qui supportent les codages de compression. Pour le moment, une servlet doit décider que sans en-tête elle ne retourne pas du contenu comprimé, ou elle doit examiner l'en-tête `User-Agent` pour savoir si le navigateur fait partie de ceux qui supportent la compression. Parmi les navigateurs les plus répandus, la plupart supportent le codage gzip, aucun ne supporte le codage compress et seuls Microsoft Internet Explorer 4 et 5 sur Windows acceptent le codage deflate. Pour déterminer facilement les possibilités d'un navigateur, y compris sa capacité à supporter GZIP, essayez le produit BrowserHawk4J de Cyscape. Ce produit commercial détecte le support de GZIP ainsi que des dizaines d'autres propriétés du navigateur dont son type, sa version, le système d'exploitation du client, les plug-ins installés, la largeur et la hauteur du navigateur, la largeur et la hauteur de l'écran du client, les cookies interdits, le JavaScript interdit et même la vitesse de connexion. Consultez le site *http://www.cyscape.com/products/bh4j*.[5]

Bien que la négociation du format de compression à utiliser puisse impliquer une quantité non négligeable de logique, envoyer réellement le contenu compressé pourrait difficilement être plus simple. La servlet ne fait qu'envelopper son `ServletOutputStream`

5. Jason est fier d'avoir été un des principaux développeurs embauchés pour créer BrowserHawk4J.

standard dans un GZIPOutputStream ou un ZipOutputStream. Soyez sûr de bien appeler out.close() lorsque la servlet a terminé d'écrire sa sortie afin que les données de fin du format de compression soient écrites. Ah, les joies de Java !

L'*exemple 6-12* montre la servlet ViewResource du chapitre 4 réécrite pour envoyer du contenu compressé chaque fois que c'est possible. Nous aurions voulu vous montrer une copie d'écran, mais il n'y a rien de nouveau à voir. Comme nous l'avons dit, un utilisateur final ne peut dire si le serveur a envoyé du contenu compressé ou non au navigateur, excepté peut-être par le temps de téléchargement réduit

Exemple 6-12. Envoyer un contenu compressé

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.zip.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class ViewResourceCompress extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    OutputStream out = null;

    // Choisir le codage de contenu adéquat en fonction de l'en-tête
    // Accept-Encoding du client. Choisir GZIP si l'en-tête contient "gzip".
    // Choisir ZIP si l'en-tête contient "compress". Sinon, pas de compression.
    // S'assurer que Content-Encoding utilise le préfixe "x-" si et seulement
    // si c'est le cas de Accept-Encoding.
    String encodings = req.getHeader("Accept-Encoding");
    if (encodings != null && encodings.indexOf("gzip") != -1) {
      // Allons-y pour GZIP.
      if (encodings.indexOf("x-gzip") != -1) {
        res.setHeader("Content-Encoding", "x-gzip");
      }
      else {
        res.setHeader("Content-Encoding", "gzip");
      }
      out = new GZIPOutputStream(res.getOutputStream());
    }
    else if (encodings != null && encodings.indexOf("compress") != -1) {
      // Allons-y pour ZIP.
      if (encodings.indexOf("x-compress") != -1) {
        res.setHeader("Content-Encoding", "x-compress");
      }
      else {
        res.setHeader("Content-Encoding", "compress");
      }
```

Exemple 6-12. Envoyer un contenu compressé (suite)

```
      out = new ZipOutputStream(res.getOutputStream());
      ((ZipOutputStream)out).putNextEntry(new ZipEntry("nom bidon"));
    }
    else {
      // Pas de compression.
      out = res.getOutputStream();
    }
    res.setHeader("Vary", "Accept-Encoding");

    // Obtenir la ressource à afficher.
    URL url = null;
    try {
      url = ServletUtils.getResource(getServletContext(), req.getPathInfo());
    }
    catch (IOException e) {
      res.sendError(
        res.SC_NOT_FOUND, "L'information de chemin supplémentaire " +
        "doit désigner une ressource valide : " + e.getMessage());
    }

    // Se connecter à la ressource.
    URLConnection con = url.openConnection();
    con.connect();

    // Obtenir et définir le type de la ressource.
    String contentType = con.getContentType();
    res.setContentType(contentType);

    // Renvoyer la ressource.
    try {
      ServletUtils.returnURL(url, out);
    }
    catch (IOException e) {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
              "Problème d'envoi de la ressource : " + e.getMessage());
    }

    // Écrire la terminaison de la compression et fermer le flot de sortie.
    out.close();
  }
}
```

La servlet commence par déclarer un `OutputStream` `null` puis par faire de cet `OutputStream` un `GZIPOutputStream`, un `ZipOutputStream` ou un `ServletOutputStream` en fonction de l'en-tête `Accept-Encoding` reçu. Lors du choix du flot de sortie à utiliser, la servlet définit l'en-tête `Content-Encoding` correspondant. Lors de l'envoi du contenu compressé, cet en-tête doit être défini pour que le client puisse exécuter le bon algorithme de décompression. La servlet définit aussi l'en-tête `Vary` à `Accept-Encoding` pour

être polie et indiquer au client que la servlet varie sa sortie en fonction de l'en-tête `Accept-Encoding`. La plupart des clients ignorent cet en-tête.

Après cette première logique, la servlet peut traiter le flot de sortie comme n'importe quel autre `OutputStream`. Elle pourrait envelopper le flot par un `PrintStream` ou un `PrintWriter`, ou elle pourrait le passer à un `GifEncoder`. Mais, indépendamment de ce qu'elle fait, la servlet doit s'assurer que l'appel à `out.close()` est bien là lorsqu'elle a fini d'envoyer le contenu. Cet appel écrit les bonnes données de fin sur le flot compressé.

Certains contenus ne doivent pas être comprimés. Par exemple, les images GIF ou JPEG sont déjà compressées, en général il n'y a aucun bénéfice à les compresser une fois de plus (et certains navigateurs ont des problèmes pour afficher des images envoyées sous forme comprimée). Une version améliorée de la servlet `FileViewCompressed` pourrait détecter l'envoi d'une image et dans ce cas éviter une compression supplémentaire.

Push du serveur

Jusqu'à présent, chaque page retournée par une servlet n'a été que cela : une page. Toujours une page avec un type de contenu. Mais pourquoi pensons-nous en des termes si limités ? Pourquoi ne pas écrire une servlet qui retournerait plusieurs pages, chacune avec un type de contenu propre, en réponse à une même requête ? C'est peut-être difficile à imaginer, et paraître encore plus difficile à réaliser, mais c'est en fait assez facile en utilisant une technique connue sous le nom de *push du serveur*.

On l'appelle ainsi car le serveur envoie (*push*) une suite de pages réponses au client. Comparez cela à la technique de pull du client utilisée dans le chapitre précédent, où c'est au client de récupérer (*pull*) chaque page sur le serveur. Bien que les résultats de chaque technique soient similaires pour l'utilisateur final, une suite de pages, les détails d'implémentation et les bonnes utilisations des deux techniques sont plutôt différents.

Avec le push du serveur, la connexion par socket entre le client et le serveur reste ouverte jusqu'à l'envoi de la dernière page. Cela donne au serveur la possibilité d'envoyer des mises à jour de pages rapidement et de contrôler exactement l'instant d'envoi de ces mises à jour. En cela, le push du serveur est idéal pour des pages qui nécessitent de fréquentes mises à jour (comme des animations rudimentaires) ou des pages qui nécessitent un contrôle du serveur mais des mises à jour sporadiques (comme des mises à jour de l'état en *live*). Cependant, notez que le push du serveur n'est pas encore supporté par Microsoft Internet Explorer et qu'une utilisation intensive doit être évitée car elle est nuisible au nombre de sockets disponibles sur le serveur.

Avec le pull du client, la connexion par socket est fermée après chaque page, la mise à jour des pages incombe donc au client. Le client utilise la valeur de l'en-tête `Refresh` envoyé par le serveur pour déterminer quand effectuer sa mise à jour. Le pull du client est donc le meilleur choix lorsque les pages ne nécessitent pas de fréquentes mises à jour ou que ces dernières se font à intervalles réguliers.

Le push du serveur peut se révéler pratique pour des animations limitées en longueur et pour des mises à jour d'état en temps réel. Par exemple, considérons une servlet devant envoyer les quatre dernières photos satellites du temps qu'il fait, créant ainsi une animation rudimentaire. Si vous vous souvenez de la servlet `PrimeSearcher` du chapitre 3, *Cycle de vie d'une servlet*, essayez d'imaginer comment nous pourrions utiliser le push

du serveur pour notifier immédiatement à un nombre limité de clients que le serveur vient de trouver un nouveau nombre premier.

L'exemple 6-13 montre une servlet qui utilise le push du serveur pour afficher le compte à rebours du lancement d'une fusée. Elle commence par envoyer une série de pages qui décomptent de 10 à 1. Chaque page remplace la page précédente. Lorsque le compteur arrive à 0, la servlet envoie une image du lancement. Elle utilise la classe `com.oreilly.servlet.MultipartResponse` (montrée à l'*exemple 6-14*) pour la gestion des détails du push du serveur.

Exemple 6-13. Compte à rebours du lancement d'une fusée

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.MultipartResponse;
import com.oreilly.servlet.ServletUtils;

public class Countdown extends HttpServlet {

  static final String LAUNCH = "/images/launch.gif";

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();  // Une sortie binaire.

    // Préparer une réponse multipartie.
    MultipartResponse multi = new MultipartResponse(res);

    // Envoyer d'abord un décompte.
    for (int i = 10; i > 0; i--) {
      multi.startResponse("text/plain");
      out.println(i + "...");
      multi.endResponse();
      try { Thread.sleep(1000); } catch (InterruptedException e) { }
    }

    // Puis envoyer l'image du lancement.
    multi.startResponse("image/gif");
    try {
      ServletUtils.returnFile(req.getRealPath(LAUNCH), out);
    } catch (FileNotFoundException e) {
      throw new ServletException("Impossible de trouver le fichier  : " +
          e.getMessage());
    }

    // Ne pas oublier la fin de la réponse multipartie.
    multi.finish();
  }
}
```

La classe `MultipartResponse` cache la plus grande partie des détails pénibles impliqués dans la gestion du push du serveur. Vous pouvez l'utiliser librement dans vos propres servlets. Elle est facile à utiliser, comme vous pouvez le constater dans l'exemple précédent.

D'abord, créez un nouvel objet `MultipartResponse`, en lui passant l'objet de réponse de la servlet. `MultipartResponse` utilise cet objet pour récupérer le flot de sortie de la servlet et pour spécifier le type de contenu de la réponse. Ensuite, pour chaque page, commencez par appeler `startResponse()` en donnant le type de contenu pour cette page. Envoyez le contenu en écrivant sur le flot de sortie. Un appel à `endResponse()` termine la page et vide le contenu afin que le client puisse le voir. À ce stade, vous pouvez ajouter un appel à `sleep()`, ou toute autre forme de délai, jusqu'à ce que la prochaine page soit prête à être envoyée. L'appel à `endResponse()` est optionnel car la méthode `startResponse()` sait si la réponse précédente est terminée et la termine si nécessaire. Vous devez appeler `endResponse()` s'il y a un délai entre le moment où une réponse termine et celui où la prochaine commence. Cela permet au client d'afficher la dernière réponse tout en attendant la prochaine. Enfin, après l'envoi de toutes les réponses, un appel à la méthode `finish()` termine la réponse multipartie et envoie un code indiquant au client qu'il n'y aura plus de réponses.

L'*exemple 6-14* contient le code de la classe `MultipartResponse`.

Exemple 6-14. La classe MultipartResponse

```
package com.oreilly.servlet;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MultipartResponse {

  HttpServletResponse res;
  ServletOutputStream out;
  boolean endedLastResponse = true;

  public MultipartResponse(HttpServletResponse response) throws IOException {
    // Sauver l'objet de réponse et le flot de sortie.
    res = response;
    out = res.getOutputStream();

    // Établir les choses.
    res.setContentType("multipart/x-mixed-replace;boundary=End");
    out.println();
    out.println("--End");
  }

  public void startResponse(String contentType) throws IOException {
    // Terminer la dernière réponse si nécessaire.
    if (!endedLastResponse) {
      endResponse();
    }
```

Exemple 6-14. La classe MultipartResponse (suite)

```
    // Démarrer la suivante.
    out.println("Content-Type: " + contentType);
    out.println();
    endedLastResponse = false;
  }

  public void endResponse() throws IOException {
    // Terminer la réponse, et vider le contenu afin que le client le voit.
    out.println();
    out.println("--End");
    out.flush();
    endedLastResponse = true;
  }

  public void finish() throws IOException {
    out.println("--End--");
    out.flush();
  }
}
```

7

Suivi de session

HTTP est un protocole sans état : il ne fournit aucun moyen à un serveur de reconnaître qu'une séquence de requêtes proviennent toutes du même client. Les défenseurs de l'anonymat pourraient considérer cela comme une fonctionnalité, mais c'est en fait générateur de problèmes car beaucoup d'applications web ne sont pas sans état. L'application de caddie est un exemple classique. Un client dépose des articles dans son caddie virtuel, les accumule jusqu'à ce qu'il les paye plusieurs requêtes de pages plus tard, ou même quelques jours plus tard !

Il est plus facile de comprendre ce problème de HTTP si l'on imagine un forum de discussion en ligne où vous êtes l'invité d'honneur. Des dizaines d'utilisateurs discutent avec vous, tous en même temps. Ils vous posent des questions, répondent à vos questions et vous font en général regretter de ne pas avoir suivi le cours de dactylographie. Maintenant, imaginez que lorsque chaque participant vous écrit, le forum de discussion ne vous indique pas qui vous parle ! Tout ce que vous voyez c'est une suite de questions et d'instructions mélangées les unes aux autres. Dans ce genre de forum, le mieux que vous puissiez faire c'est de maintenir des conversations simples, peut-être en répondant aux questions directes. Si vous essayez de faire un peu plus, comme poser une question à quelqu'un, vous ne saurez pas nécessairement lorsque la réponse va revenir. C'est exactement le problème d'état de HTTP. Le serveur HTTP voit uniquement une série de requêtes, il a besoin d'une aide supplémentaire pour savoir exactement qui fait la requête[1].

La solution, comme vous l'avez sans doute deviné, est que le client se présente lui-même à chaque requête. Chaque client doit fournir un identifiant unique permettant au serveur de l'identifier, ou il doit donner des informations que le serveur peut utiliser pour traiter correctement la requête. En reprenant l'exemple du forum de discussion, un participant doit commencer chacune de ses phrases par quelque chose comme « Bonjour,

1. Si vous vous demandez pourquoi le serveur HTTP ne peut identifier le client par l'adresse IP de la machine qui se connecte, la réponse est que l'adresse IP donnée peut correspondre à l'adresse d'un serveur proxy ou à celle d'une machine serveur hébergeant de multiples utilisateurs.

c'est Jason, et... » ou « Bonjour, je viens juste de te demander ton âge, et... ». Comme vous le verrez dans ce chapitre, les clients HTTP ont plusieurs possibilités pour envoyer cette information avec chaque requête.

La première moitié de ce chapitre explore les techniques de suivi de session classiques utilisées par les développeurs CGI : authentification de l'utilisateur, champs de formulaire cachés, réécriture d'URL et cookies persistants. La seconde moitié explique le support intégré pour le suivi de session de l'API Servlet. Ce support est construit au-dessus des techniques traditionnelles et simplifie grandement le suivi de session dans vos servlets. Tout ce chapitre suppose l'utilisation d'un seul serveur. Le chapitre 12, *Servlets d'entreprise et J2EE*, explique comment gérer l'état de sessions partagées sur plusieurs serveurs.

Authentifier l'utilisateur

Une façon d'effectuer le suivi de session est d'exploiter les informations venant avec l'authentification de l'utilisateur. Nous avons parlé de l'authentification de l'utilisateur au chapitre 4, *Retrouver l'information*, mais, au cas où vous l'auriez oublié, elle survient lorsqu'un serveur web restreint l'accès à certaines de ses ressources aux clients qui se connectent en utilisant un nom et un mot de passe reconnus. Après la connexion du client, le nom de l'utilisateur est disponible pour la servlet via `getRemoteUser()`.

Nous pouvons utiliser le nom de l'utilisateur pour suivre la session d'un client. Une fois l'utilisateur connecté, le navigateur se souvient de son nom et le renvoie avec le mot de passe lorsque l'utilisateur visite de nouvelles pages. Une servlet peut identifier l'utilisateur par son nom et donc suivre sa session. Par exemple, si l'utilisateur ajoute des articles dans son caddie virtuel, il est possible de s'en souvenir (peut-être dans une classe partagée ou une base de données externe) et de l'utiliser plus tard dans une autre servlet lorsque cet utilisateur va aller à la page de paiement.

Par exemple, une servlet qui utilise l'authentification de l'utilisateur pourrait ajouter un article dans le caddie de l'utilisateur grâce à un code similaire au suivant :

```
String name = req.getRemoteUser();
if (name == null) {
  // Expliquer que l'administrateur du serveur doit protéger cette page.
} else {
  String[] items = req.getParameterValues("item");
  if (items != null) {
    for (int i = 0; i < items.length; i++) {
      addItemToCart(name, items[i]);
    }
  }
}
```

Une autre servlet peut ensuite retrouver les articles du caddie de l'utilisateur avec le code suivant :

```
String name = req.getRemoteUser();
if (name == null) {
  // Expliquer que l'administrateur du serveur doit protéger cette page.
}
```

```
else {
  String[] items = getItemsFromCart(name);
}
```

Le plus gros avantage de l'emploi de l'authentification de l'utilisateur pour le suivi de session est que cette technique est simple à mettre en œuvre. Il suffit d'indiquer au serveur de protéger un ensemble de pages (en suivant les instructions données au chapitre 8, *Sécurité*) et d'utiliser `getRemoteUser()` pour identifier chaque client. Un autre avantage est qu'elle fonctionne bien même lorsque l'utilisateur accède au site à partir de machines différentes. Elle fonctionne aussi même si l'utilisateur quitte votre site ou sort de son navigateur avant d'y revenir.

Le plus gros inconvénient de l'authentification de l'utilisateur est qu'elle impose que chaque utilisateur s'enregistre et qu'ensuite il se connecte chaque fois qu'il visite votre site. La plupart des utilisateurs accepteront cet enregistrement et cette connexion lorsqu'ils accéderont à des informations sensibles, mais c'est beaucoup trop pour un simple suivi de session. Un autre inconvénient est l'authentification de base de HTTP n'offre aucun mécanisme de fermeture de session ; l'utilisateur doit fermer son navigateur pour se déconnecter. Le dernier problème de l'authentification de l'utilisateur est que l'utilisateur ne peut maintenir en parallèle plus d'une session sur le même site. Nous avons clairement besoin d'autres approches pour supporter un suivi de session anonyme et pour supporter un suivi de session authentifiée avec fermeture de session.

Champs de formulaire cachés

Une façon de supporter le suivi de session anonyme est d'utiliser les champs de formulaire cachés. Comme le nom l'indique, ces champs ajoutés à un formulaire HTML ne sont pas affichés dans le navigateur du client. Ils sont passés au serveur lorsque le formulaire qui les contient est renvoyé. Vous pouvez inclure des champs cachés de la façon suivante :

```
<FORM ACTION="/servlet/MovieFinder" METHOD="POST">
...
<INPUT TYPE=hidden NAME="zip" VALUE="94040">
<INPUT TYPE=hidden NAME="level" VALUE="expert">
...
</FORM>
```

D'une certaine manière, les champs de formulaire cachés définissent des constantes pour le formulaire. Pour une servlet qui reçoit le formulaire envoyé, il n'y a pas de différence entre un champ caché et un champ visible.

Avec les champs de formulaire cachés, vous pouvez réécrire vos servlets de caddie de manière à ce que les utilisateurs puissent faire leurs courses de façon anonyme jusqu'au moment du paiement. L'*exemple 7-1* montre l'utilisation de cette technique avec une servlet affichant le contenu du caddie de l'utilisateur et permettant à l'utilisateur d'ajouter d'autres articles ou de payer. Un écran d'exemple pour un lecteur assidu est montré à la *figure 7-1*.

Exemple 7-1. Suivi de session utilisant les champs de formulaire cachés

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerHidden extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HEAD><TITLE>Articles dans le caddie</TITLE></HEAD>");
    out.println("<BODY>");

    // Les articles du caddie sont passés dans le paramètre item.
    String[] items = req.getParameterValues("item");

    // Afficher les articles du caddie.
    out.println("Voici les articles qui se trouvent dans votre caddie :<BR>");
    if (items == null) {
      out.println("<B>Aucun</B>");
    }
    else {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++) {
        out.println("<LI>" + items[i]);
      }
      out.println("</UL>");
    }

    // Demander si l'utilisateur veut ajouter d'autres articles ou payer. Les
    // articles courants sont mis dans des champs cachés afin de les passer.
    out.println("<FORM ACTION=\"/servlet/ShoppingCart\" METHOD=POST>");
    if (items != null) {
      for (int i = 0; i < items.length; i++) {
        out.println("<INPUT TYPE=HIDDEN NAME=\"item\" VALUE=\"" +
          items[i] + "\">");
      }
    }
    out.println("Voulez-vous <BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Ajouter d'autres articles \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Passer au paiement \">");
    out.println("</FORM>");

    out.println("</BODY></HTML>");
  }
}
```

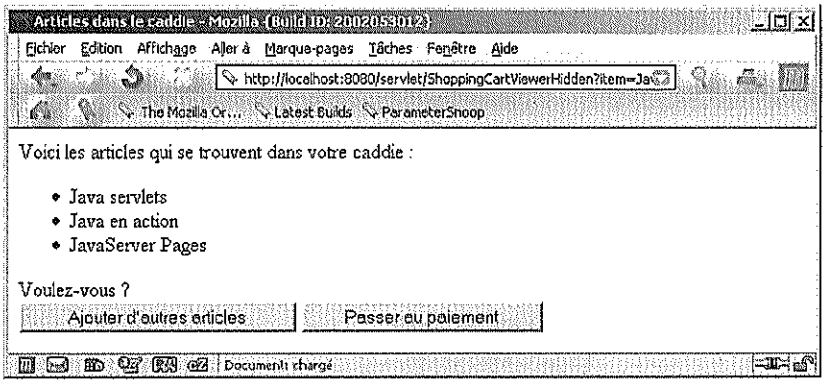


Figure 7-1. Contenu du caddie

Cette servlet commence par lire les articles déjà dans le caddie en utilisant `getParameterValues("item")`. Les valeurs du paramètre `item` ont été envoyées à cette servlet en utilisant des champs cachés. La servlet montre ensuite la liste des articles courants à l'utilisateur et lui demande s'il veut en ajouter d'autres ou passer au paiement. La question est posée avec un formulaire qui contient des champs cachés afin que la cible du formulaire (la servlet `ShoppingCart`) reçoive les articles courants dans l'envoi.

Au fur et à mesure que plus d'information est associée à la session du client, il devient lourd de la passer en utilisant des champs cachés. Dans ce cas, il est possible de ne passer qu'un identifiant de session unique permettant d'identifier une session cliente particulière. Cet identifiant de session peut être associé à l'information complète sur la session stockée sur le serveur.

Sachez que les identifiants de session doivent rester secrets car tout client connaissant l'identifiant de session d'un autre client peut, avec un champ de formulaire caché qu'il aura créé, prendre l'identité du second client. C'est pourquoi, les identifiants de session doivent être générés de manière à ce qu'ils soient difficiles à deviner ou à créer. De plus les identifiants de session active doivent être protégés. Par exemple, il ne faut pas que le journal du serveur soit publiquement accessible car les URL journalisées peuvent contenir les identifiants de session pour les formulaires envoyés par des requêtes GET.

Les champs de formulaires cachés peuvent servir à implémenter l'authentification avec fermeture de session. Il suffit de présenter un formulaire d'ouverture de session et, une fois l'utilisateur authentifié par le serveur, d'associer son identité à son identifiant de sessions particulier. Pour fermer la session, l'identifiant est détruit (en ne l'envoyant plus au client dans les formulaires suivants) ou l'association entre l'identifiant et l'utilisateur est simplement oubliée. Le chapitre 8 détaille cette approche.

Les avantages des champs de formulaire cachés sont leur ubiquité et le support de l'anonymat. Les champs cachés sont supportés par tous les navigateurs, ils ne demandent aucune particularité sur le serveur et ils peuvent être utilisés avec les clients qui ne se sont pas enregistrés. Le principal inconvénient de cette technique est qu'elle ne fonctionne que pour une séquence de formulaires générés dynamiquement. La session ne peut être maintenue avec les documents statiques, les documents envoyés par e-mail, les documents ajoutés aux signets et les arrêts du navigateur.

Réécrire une URL

La réécriture d'URL est une autre façon de supporter le suivi de session anonyme. Avec cette technique, chaque URL locale sur laquelle peut cliquer un utilisateur est modifiée dynamiquement, ou réécrite, afin d'inclure des informations supplémentaires. Ces informations peuvent prendre la forme d'une information de chemin supplémentaire, de paramètres ajoutés, ou d'un changement d'URL propre au serveur. Du fait de la place disponible limitée pour la réécriture d'une URL, l'information supplémentaire est habituellement limitée à un identifiant de session unique. Par exemple, les URL suivantes ont été réécrites pour passer l'identifiant de session `123` :[2]

`http://serveur:port/servlet/Rewritten`	originale
`http://serveur:port/servlet/Rewritten/123`	information de chemin supplémentaire
`http://serveur:port/servlet/Rewritten?sessionid=123`	paramètre ajouté
`http://serveur:port/servlet/Rewritten;jsessionid=123`	modification personnalisée

Chaque technique de réécriture a ses avantages et ses inconvénients. L'information de chemin supplémentaire fonctionne sur tous les serveurs, mais elle ne fonctionne pas bien si une servlet doit utiliser l'information de chemin supplémentaire comme une information de chemin réel. Les paramètres ajoutés fonctionnent aussi sur tous les serveurs, mais ils peuvent créer des collisions de noms de paramètres. La modification propre au serveur fonctionne dans toutes les conditions pour les serveurs qui supportent la modification. Malheureusement, cela ne fonctionne pas du tout pour les serveurs qui ne l'accepte pas.

L'*exemple 7-2* montre une version revue de notre visualiseur de caddie qui utilise la réécriture d'URL par information de chemin supplémentaire pour suivre de façon anonyme une session de caddie

Exemple 7-2. Suivi de session utilisant la réécriture d'URL

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerRewrite extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HEAD><TITLE>Articles dans le caddie</TITLE></HEAD>");
    out.println("<BODY>");
```

2. Rappelez-vous que des identifiants de session doivent être difficiles à deviner ou à créer. On ne peut pas dire que `123` soit un bon identifiant de session, excepté pour un simple exemple dans un livre.

Exemple 7-2. Suivi de session utilisant la réécriture d'URL (suite)

```
    // Obtenir l'identifiant de la session courante, ou en générer un.
    String sessionid = req.getPathInfo();
    if (sessionid == null) {
      sessionid = generateSessionId();
    }

    // Les articles du caddie sont associés à l'identifiant de session.
    String[] items = getItemsFromCart(sessionid);

    // Afficher les articles du caddie.
    out.println("Voici les articles qui se trouvent dans votre caddie :<BR>");
    if (items == null) {
      out.println("<B>Aucun</B>");
    }
    else {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++) {
        out.println("<LI>" + items[i]);
      }
      out.println("</UL>");
    }

    // Demander si l'utilisateur veut ajouter d'autres articles ou payer.
    // Inclure l'identifiant de session dans l'URL d'action.
    out.println("<FORM ACTION=\"/servlet/ShoppingCart/" + sessionid +
                "\" METHOD=POST>");
    out.println("Voulez-vous ?<BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Ajouter d'autres articles \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Passer au paiement \">");
    out.println("</FORM>");

    // Proposer une page d'aide. Inclure l'identifiant de session dans l'URL.
    out.println("Pour de l'aide, cliquez <A HREF=\"/servlet/Help/" + sessionid +
                "?topic=ShoppingCartViewerRewrite\">ici</A>");

    out.println("</BODY></HTML>");
  }

  private static String generateSessionId() {
    String uid = new java.rmi.server.UID().toString();  // Garanti unique.
    return java.net.URLEncoder.encode(uid); // Coder les caractères spéciaux.
  }

  private static String[] getItemsFromCart(String sessionid) {
    // Non implémentée.
  }
}
```

Cette servlet essaie d'abord de retrouver l'identifiant de session courante en utilisant `getPathInfo()`. Si l'identifiant n'est pas spécifié, elle appelle `generateSessionId()` pour générer un nouvel identifiant de session unique en utilisant une classe RMI conçue spécifiquement pour cela. L'identifiant de session est utilisé pour récupérer et afficher les articles courants dans le caddie. L'identifiant est ensuite ajouté à l'attribut `ACTION` du formulaire afin qu'il puisse être retrouvé par la servlet `ShoppingCart`. L'identifiant de session est aussi ajouté à une nouvelle URL d'aide qui invoque la servlet `Help`. Cela n'était pas possible avec les champs de formulaire cachés car la servlet `Help` n'est pas la cible du formulaire.

Les avantages et les inconvénients de la réécriture d'URL correspondent à ceux des champs de formulaire cachés. Ces deux techniques fonctionnent avec tous les navigateurs, elles autorisent les accès anonymes et elles peuvent servir à implémenter une authentification avec fermeture de session. La principale différence est que la réécriture d'URL fonctionne pour tous les documents créés dynamiquement, comme la servlet `Help`, et pas seulement les formulaires. De plus, avec le support adéquat du serveur, la réécriture d'URL personnalisée peut même fonctionner pour les documents statiques. Malheureusement, la réécriture d'URL peut vite devenir pénible.

Cookies persistants

Une quatrième technique pour effectuer le suivi de session implique les cookies persistants. Un cookie est une information envoyée à un navigateur par un serveur web qui peut ensuite être relue par ce navigateur. Lorsqu'un navigateur reçoit un cookie, il le sauve et le renvoie ensuite au serveur chaque fois qu'il accède à une page sur ce serveur, conformément à certaines règles. La valeur d'un cookie pouvant identifier de façon unique un client, ils sont souvent utilisés pour le suivi de session.

Les cookies ont été introduits la première fois dans Netscape Navigator. Bien qu'ils ne faisaient pas partie de la spécification HTTP officielle, ils sont rapidement devenus un standard *de facto* supporté dans tous les navigateurs y compris Netscape 0.94 Beta et suivants, ainsi que Microsoft Internet Explorer 2 et suivants. Aujourd'hui, le HTTP Working Group de l'IETF (*Internet Engineering Task Force*) est en train d'officialiser les cookies comme il est écrit dans la RFC 2109. Pour plus d'informations sur les cookies, allez voir la spécification des cookies de Netscape à *http://home.netscape.com/newsref/std/cookie_spec.html* et la RFC 2109 à *http://www.ietf.org/rfc/rfc2109.txt*. Un autre site intéressant est *http://www.cookie-central.com*.

Utiliser les cookies

L'API Servlet fournit la classe `javax.servlet.http.Cookie` pour travailler avec les cookies. Les détails de l'en-tête HTTP pour les cookies sont gérés par l'API Servlet. Vous créez un cookie avec le constructeur `Cookie()` :

```
public Cookie(String name, String value)
```

Cela crée un nouveau cookie avec un nom initial et une valeur. Les règles de validité des noms et des valeurs sont données dans la spécification des cookies de Netscape et la RFC 2109.

Une servlet peut envoyer un cookie à un client en passant un objet `Cookie` à la méthode `addCookie()` de `HttpServletResponse` :

```
public void HttpServletResponse.addCookie(Cookie cookie)
```

Cette méthode ajoute le cookie spécifié à la réponse. D'autres cookies peuvent être ajoutés par d'autres appels à `addCookie()`. Les cookies étant envoyés en utilisant les en-têtes HTTP, ils doivent être ajoutés à la réponse avant que la réponse ne soit validée. La seule contrainte sur les navigateurs est qu'ils acceptent 20 cookies par site, 300 par utilisateur et ils peuvent limiter la taille de chaque cookie à 4 096 octets.

Voici le code pour créer un cookie :

```
Cookie cookie = new Cookie("ID", "123");
res.addCookie(cookie);
```

Une servlet retrouve les cookies en appelant la méthode `getCookies()` de `HttpServletRequest` :

```
public Cookie[] HttpServletRequest.getCookies()
```

Cette méthode retourne un tableau d'objets `Cookie` qui contient tous les cookies envoyés par le navigateur dans la requête ou un tableau vide si aucun cookie n'a été envoyé. Dans les versions de l'API Servlet antérieures à la version 2.1, cette méthode retournait `null` si aucun cookie n'avait été envoyé. Pour une meilleure portabilité, il est préférable de supposer que la valeur de retour est, soit `null`, soit un tableau vide.

Le code pour récupérer les cookies ressemble au suivant :

```
Cookie[] cookies = req.getCookies();
if (cookies != null) {
  for (int i = 0; i < cookies.length; i++) {
    String name = cookies[i].getName();
    String value = cookies[i].getValue();
  }
}
```

Malheureusement, il n'existe pas de méthode dans l'API Servlet standard permettant de récupérer la valeur d'un cookie par son nom. Vous pouvez définir un certain nombre d'attributs du cookie en plus de son nom et de sa valeur. Les méthodes suivantes sont utilisées pour définir ces attributs. Comme vous pouvez le voir dans l'annexe B, *L'API Servlet HTTP, référence rapide*, il existe une méthode `get` correspondante à chaque méthode `set`. Cependant, les méthodes `get` sont rarement utilisées car lorsqu'un cookie est envoyé au serveur, il ne contient que son nom, sa valeur et sa version. Si vous définissez un attribut sur un cookie reçu d'un client, vous devez l'ajouter à la réponse pour que les modifications soient prises en compte. Vous devez aussi faire attention que tous les attributs, sauf le nom, la valeur et la version, soient aussi remis à zéro.

```
public void Cookie.setVersion(int v)
```

définit la version d'un cookie. Les servlets peuvent envoyer et recevoir des cookies formatés conformément aux cookies persistants de Netscape (version 0) ou à ceux plus récents, quelque peu expérimentaux, de la RFC 2109 (version 1). Les cookies nouvellement construits correspondent à la version 0 par défaut afin d'optimiser l'interopérabilité.

```
public void Cookie.setDomain(String pattern)
```

spécifie un motif de restriction de domaine. Un motif de domaine précise les serveurs

qui doivent voir un cookie. Par défaut, les cookies ne sont retournés qu'à la machine qui les a sauvés. En spécifiant un motif de nom de domaine on contourne cela. Le motif doit commencer par un point et contenir au moins deux points. Un motif ne correspond qu'à une seule entrée après le point initial. Par exemple, *.foo.com* est valide et correspond à *www.foo.com* et *upload.foo.com* mais pas à *www.upload.foo.com*.[3] Pour plus de détails sur les motif de domaine, reportez-vous à la spécification des cookies de Netscape et à la RFC 2109.

```
public void Cookie.setMaxAge(int expiry)
```

spécifie l'âge maximum du cookie, en secondes, avant qu'il n'expire. Une valeur négative indique que le cookie doit expirer lorsque le navigateur se termine (par défaut). Une valeur 0 indique au navigateur de supprimer le cookie immédiatement.

```
public void Cookie.setPath(String uri)
```

spécifie un chemin pour le cookie, qui est un sous-ensemble des URI auxquelles un cookie peut être envoyé. Par défaut, les cookies sont envoyés à la page qui définit le cookie et à toutes les pages du répertoire et en dessous. Par exemple, si */servlet/CookieMonster* définit un cookie, la chemin par défaut est */servlet*. Ce chemin indique que le cookie doit être envoyé à */servlet/Elmo* et à */servlet/subdir/BigBird*, mais pas à l'alias de servlet */Oscar.html* ou tout autre programmes CGI sous */cgi-bin*. Un chemin mis à / fait que le cookie est envoyé à toutes les pages du serveur. Un chemin de cookie doit être défini de façon à inclure la servlet qui a défini le cookie.

```
public void Cookie.setSecure(boolean flag)
```

indique si le cookie doit être envoyé *via* un canal sécurisé, comme SSL. Par défaut cette valeur vaut `false`.

```
public void Cookie.setComment(String comment)
```

définit le champ de commentaire du cookie. Un commentaire décrit l'objectif d'un cookie. Les navigateurs web peuvent décider d'afficher ce texte à l'utilisateur. Les commentaires ne sont pas supportés dans la version 0.

```
public void Cookie.setValue(String newValue)
```

affecte une nouvelle valeur au cookie. Avec les cookies de la version 0, les valeurs ne doivent pas contenir les caractères suivants : espaces, crochets, parenthèses, signes égal, virgules, guillemets, barres obliques, points d'interrogations, signes @, deux-points et points-virgules. Des valeurs vides ne se comportent pas de la même façon sur tous les navigateurs.

---

3. Techniquement, d'après la spécification des cookies de Netscape, deux points sont nécessaires pour les domaines de haut niveau de *.com*, *.edu*, *.net*, *.org*, *.gov*, *.mil* et *.int*, tandis que trois points sont nécessaires pour tous les autres domaines. Cette règle empêche, par exemple, un serveur de définir accidentellement ou malicieusement un cookie pour le domaine *.co.uk* et de le passer à toutes les sociétés du Royaume Uni. Malheureusement, quasiment tous les navigateurs ignorent la règle des trois points ! Pour plus d'informations, voir *http://homepages.paradise.net.nz/~glineham/cookiemonster.html*.

Faire ses courses avec des cookies persistants

L'*exemple 7-3* montre une version de notre visualiseur de caddie qui a été modifié pour utiliser les cookies persistants.

Exemple 7-3. Suivi de session utilisant les cookies persistants

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerCookie extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir l'identifiant de session courante en cherchant dans les
    // cookies reçus.
    String sessionid = null;
    Cookie[] cookies = req.getCookies();
    if (cookies != null) {
      for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("sessionid")) {
          sessionid = cookies[i].getValue();
          break;
        }
      }
    }

    // Si l'identifiant de session n'a pas été envoyé, le générer.
    // S'assurer de l'envoyer au client avec la réponse.
    if (sessionid == null) {
      sessionid = generateSessionId();
      Cookie c = new Cookie("sessionid", sessionid);
      res.addCookie(c);
    }

    out.println("<HEAD><TITLE>Articles dans le caddie</TITLE></HEAD>");
    out.println("<BODY>");

    // Les articles du caddie sont associés à l'identifiant de session.
    String[] items = getItemsFromCart(sessionid);

    // Afficher les articles du caddie.
    out.println("Voici les articles qui se trouvent dans votre caddie :<BR>");
    if (items == null) {
      out.println("<B>Aucun</B>");
    }
```

Exemple 7-3. Suivi de session utilisant les cookies persistants (suite)

```
    else {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++) {
        out.println("<LI>" + items[i]);
      }
      out.println("</UL>");
    }

    // Demander à l'utilisateur s'il veut ajouter d'autres articles ou payer.
    out.println("<FORM ACTION=\"/servlet/ShoppingCart\" METHOD=POST>");
    out.println("Voulez-vous ?<BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Ajouter d'autres articles \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Passer au paiement \">");
    out.println("</FORM>");

    // Proposer une page d'aide.
    out.println("Pour de l'aide, cliquez <A HREF=\"/servlet/Help" +
                "?topic=ShoppingCartViewerCookie\">ici</A>");

    out.println("</BODY></HTML>");
  }

  private static String generateSessionId() {
    String uid = new java.rmi.server.UID().toString();  // Garanti unique.
    return java.net.URLEncoder.encode(uid); // Coder les caractères spéciaux.
  }

  private static String[] getItemsFromCart(String sessionid) {
    // Non implémentée.
  }
}
```

Cette servlet commence par essayer de récupérer l'identifiant de session du client en itérant sur tous les cookies qu'elle reçoit avec la requête. Si aucun cookie ne contient d'identifiant de session, la servlet en génère un nouveau avec `generateSessionId()` et ajoute à la réponse un cookie contenant le nouvel identifiant de session. Le reste de cette servlet est équivalent à la version par réécriture d'URL, excepté qu'elle n'effectue pas de ré-écriture.

Les cookies persistants offrent une façon élégante, efficace et facile d'implémenter le suivi de session. Ils fournissent une introduction à chaque requête de la façon la plus automatique possible que vous pourriez souhaiter. Pour chaque requête, un cookie peut automatiquement fournir un identifiant de session client ou peut-être une liste des préférences du client. De plus, la personnalisation des cookies leur donne plus de puissance et de possibilités.

Le plus gros problème des cookies est que les navigateurs ne les acceptent pas toujours. Quelquefois, c'est parce que le navigateur ne supporte pas les cookies. Le plus souvent, c'est parce que l'utilisateur a configuré son navigateur pour qu'il refuse les cookies (pas

forcément pour des problèmes de vie privée). Les périphériques WAP n'acceptent pas non plus les cookies du fait de leur espace mémoire limité. Et ce n'est que depuis peu que les passerelles WAP commencent à gérer les cookies pour le compte des périphériques. Si un de vos clients peut ne pas accepter les cookies, vous devez alors vous rabattre sur les solutions présentées précédemment dans ce chapitre.

L'API de suivi de session

Heureusement pour nous, développeurs de servlets, il n'est pas toujours nécessaire qu'une servlet gère ses propres sessions en utilisant les techniques que nous venons de décrire. L'API Servlet fournit plusieurs méthodes et classes conçues spécifiquement pour gérer le suivi de session pour le compte de la servlet. Autrement dit, les servlets intègrent le suivi de session.[4]

L'API Session Tracking, comme nous appelons la portion de l'API Servlet dédiée au suivi de session, doit être supportée dans tout serveur web qui supporte les servlets. Cependant, le niveau de support dépend du serveur. La plupart des serveurs supportent le suivi de session à l'aide des cookies persistants, avec la possibilité de se rabattre sur la réécriture d'URL quand les cookies échouent. Certains serveurs sont capables d'écrire sur leur disque des objets de session lorsque la mémoire se remplit ou lorsque le serveur s'arrête. (Les éléments placés dans la session doivent implémenter l'interface `Serializable` pour tirer parti de cette option.) Reportez-vous à la documentation de votre serveur pour les détails propres à votre serveur. Le reste de cette section décrit le plus petit dénominateur commun des fonctionnalités fournies par l'API Servlet 2.2 sur un serveur non distribué. Rappelez-vous, le chapitre 12 présente le suivi de session distribué.

Les bases du suivi de session

Le suivi de session est merveilleusement élégant. Chaque utilisateur d'un site est associé à un objet `javax.servlet.http.HttpSession` que les servlets utilisent pour stocker ou retrouver des informations sur l'utilisateur. Vous pouvez sauver tout ensemble d'objets Java arbitraires dans un objet de session. Par exemple, un objet de session d'utilisateur fournit un emplacement pratique à une servlet pour stocker le contenu du caddie de l'utilisateur, ou, comme nous le verrons au chapitre 9, *Se connecter aux bases de données*, sa connexion à la base de données.

Pour qu'une application web opère de façon indépendante, les sessions se trouvent au niveau de la portée de l'application web. Cela signifie que chaque `ServletContext` maintient sa propre réserve d'instances de `HttpSession` et une servlet qui s'exécute dans un contexte ne peut accéder aux informations de session sauvegardées par une servlet d'un autre contexte.

---

4. Oui, nous nous sentons un peu comme l'instituteur du primaire qui vous a enseigné toutes les étapes d'une longue division, pour ne vous révéler que plus tard comment utiliser une calculatrice pour faire la même chose. Mais nous pensons, comme l'a probablement pensé votre instituteur, que vous comprendrez mieux les concepts après avoir d'abord appris l'approche traditionnelle.

Une servlet utilise la méthode getSession() de son objet de requête pour retrouver l'objet HttpSession courant :

```
public HttpSession HttpServletRequest.getSession(boolean create)
```

Cette méthode retourne la session courante associée à l'utilisateur effectuant la requête. Si l'utilisateur n'a pas de session valide, cette méthode en crée une si create vaut true ou retourne null si create vaut false. L'utilisation standard de la méthode est getSession(true). C'est pourquoi, l'API Servlet 2.1 a introduit une version sans argument qui suppose que le paramètre create vaut true :

```
public HttpSession HttpServletRequest.getSession()
```

Pour garantir que la session est correctement maintenue, la méthode getSession() doit être appelée au moins une fois avant de valider la réponse.

L'ajout de données à un objet HttpSession se fait avec la méthode setAttribute() :

```
public void HttpSession.setAttribute(String name, Object value)
```

Cette méthode lie la valeur de l'objet spécifiée au nom spécifié. Tout lien de même nom existant est remplacé. Pour retrouver un objet d'une session, utilisez getAttribute() :

```
public Object HttpSession.getAttribute(String name)
```

Cette méthode retourne l'objet lié au nom précisé ou null s'il n'y a pas de lien. Vous pouvez aussi obtenir le nom de tous les objets liés à une session avec getAttributeNames() :

```
public Enumeration HttpSession.getAttributeNames()
```

Cette méthode retourne un Enumeration qui contient les noms de tous les objets liés à cette session sous forme d'objets String ou un Enumeration vide s'il n'y a pas de lien. Enfin, vous pouvez supprimer un objet d'une session avec removeAttribute() :

```
public void HttpSession.removeAttribute(String name)
```

Cette méthode supprime l'objet lié sous le nom précisé ou ne fait rien s'il n'y a pas de lien. Chacune de ces méthodes peut lancer une java.lang.IllegalStateException si la session accédée est invalide (nous expliquerons les sessions invalides dans une prochaine section).

Notez que l'API Servlet 2.2 a changé le nom de ces méthodes, de setValue(), getValue(), getValueNames() et removeValue() en des noms plus standard, setAttribute(), getAttribute(), getAttributeNames() et removeAttribute(). Les méthodes Value sont toujours là mais leur utilisation a été désapprouvée.

Un compteur de hits utilisant le suivi de session

L'*exemple 7-4* montre une servlet simple qui utilise le suivi de session pour compter le nombre de fois qu'un client y a accédé (voir *figure 7-2*). La servlet affiche aussi toutes les liaisons de la session courante, simplement parce qu'elle peut le faire.

Exemple 7-4. Suivi de session pour un compteur de hits

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
```

Exemple 7-4. Suivi de session pour un compteur de hits (suite)

```
import javax.servlet.http.*;

public class SessionTracker extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir l'objet de session courante, en créer un si nécessaire.
    HttpSession session = req.getSession();

    // Incrémenter le compteur de hits pour cette page. La valeur est sauvée
    // dans la session du client sous le nom "tracker.count".
    Integer count = (Integer)session.getAttribute("tracker.count");
    if (count == null)
      count = new Integer(1);
    else
      count = new Integer(count.intValue() + 1);
    session.setAttribute("tracker.count", count);

    out.println("<HTML><HEAD><TITLE>SessionTracker</TITLE></HEAD>");
    out.println("<BODY><H1>Démonstration du suivi de session</H1>");

    // Afficher le compteur de hits pour cette page.
    out.println("Vous avez visité cette page " + count + " fois.");

    out.println("<P>");

    out.println("<H2>Voici les données de votre session :</H2>");
    Enumeration enum = session.getAttributeNames();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      out.println(name + " : " + session.getAttribute(name) + "<BR>");
    }
    out.println("</BODY></HTML>");
  }
}
```

Cette servlet récupère d'abord l'objet `HttpSession` associé au client courant. En passant `true` à `getSession()`, elle demande la création d'une session si nécessaire. La servlet obtient ensuite un objet `Integer` lié au nom `tracker.count`. Si cet objet n'existe pas, la servlet démarre un nouveau compteur. Sinon, elle remplace l'`Integer` par un nouveau dont la valeur a été incrémentée de 1. Enfin, la servlet affiche le compteur courant et toutes les paires nom/valeur de la session.

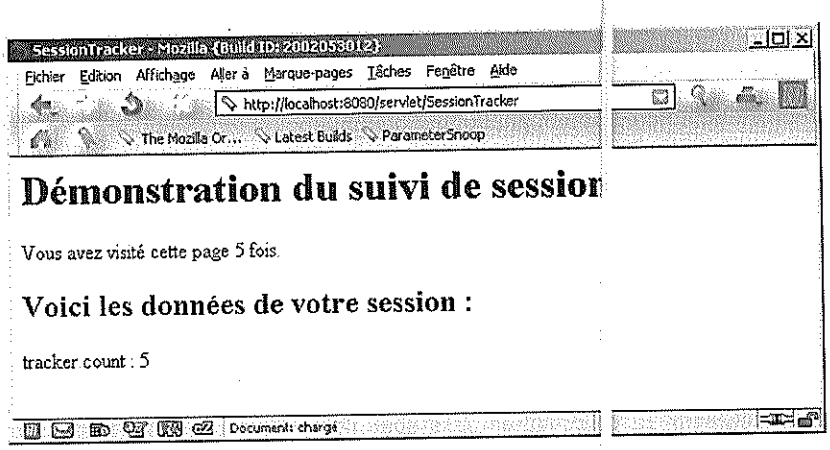


Figure 7-2. Compter les visites du client

Cycle de vie d'une session

Les sessions ne durent pas indéfiniment. Une session expire, soit automatiquement après un certain temps d'inactivité (30 minutes par défaut pour le serveur Tomcat), soit manuellement lorsqu'elle est explicitement invalidée par un servlet. Un arrêt de la servlet peut ou non invalider une session, en fonction des possibilités du serveur. Lorsque la session expire (ou est invalidée), l'objet `HttpSession` et les données qu'il contient sont supprimés du système.

Attention, toute information sauvée dans un objet de session d'utilisateur est perdue lorsque la session est invalidée. Si vous devez conserver les informations, il vous faut les garder dans un autre endroit (comme une base de données) et fournir votre propre mécanisme associant l'utilisateur avec ses entrées dans la base de données. Le plus souvent, cela se fait en demandant une certaine forme d'ouverture de session, en définissant un cookie à durée de vie longue pouvant être lu pendant des années et/ou en donnant à l'utilisateur une URL réécrite qu'il peut placer dans ses signets.

Mon Yahoo! est un exemple du monde réel. Les utilisateurs ouvrent une session *via* un formulaire HTML, le suivi de session se fait à l'aide d'un cookie qui expire lorsque le navigateur est fermé et (s'ils cochent la case Mémoriser compte et mot de passe) ils obtiennent un cookie persistant qui enregistre leur identité jusqu'à ce qu'ils ferment explicitement la session. Il est particulièrement intéressant que même si Yahoo peut connaître l'identité de l'utilisateur grâce au cookie persistant il faut que l'utilisateur entre son mot de passe avant d'accéder à l'e-mail et aux autres informations sensibles.

Le suivi de session intégré peut toujours être utilisé avec des sessions longues pour gérer la session de connexion, stocker une référence vers des données externes et/ou détenir une copie d'informations de la base de données afin d'accélérer les recherches.

Fixer le timeout d'une session

Idéalement, une session doit devenir invalide dès que l'utilisateur ferme son navigateur, passe à un site différent ou s'éloigne de son bureau. Malheureusement, un serveur ne dispose d'aucun moyen pour détecter ces événements. Par conséquent, les sessions pas-

sent par une certaine période d'inactivité après que le serveur a supposé que l'utilisateur est parti et qu'il n'est plus nécessaire de maintenir son état de session.

Le timeout de session par défaut peut être indiqué dans le descripteur de déploiement *web.xml* ; il s'applique à toutes les sessions créées dans l'application web donnée. Voir l'*exemple 7-5*.

Exemple 7-5. Fixer le timeout par défaut à une heure

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <!-- ..... -->
    <session-config>
        <session-timeout>
            60 <!-- minutes -->
        </session-timeout>
    </session-config>
</web-app>
```

La balise `<session-timeout>` contient une valeur de timeout donnée en minutes. Dans cet exemple, nous indiquons que si l'utilisateur n'a effectué aucune requête sur cette application web pendant 60 minutes, le serveur peut invalider sa session et délier les objets qu'elle contient. La spécification des servlets impose que les valeurs soient des « nombres entiers », les valeurs négatives étant exclues, mais certains serveurs utilisent une valeur négative pour indiquer que les sessions ne doivent jamais expirer.

Il est aussi possible de donner des valeurs de timeout pour chaque session individuelle. L'objet `HttpSession` offre pour cela la méthode `setMaxInactiveInterval()` :

```
public void HttpSession.setMaxInactiveInterval(int secs)
```

Cette méthode spécifie la valeur de timeout pour la session, donnée en *secondes*. Une valeur négative indique que la sessions ne doit jamais expirer. Les unités ne correspondent pas à celles de la balise `<session-timeout>` ; il n'y a pas de bonne raison à cela, excepté l'accident. Voici une petite astuce pour se souvenir quelle unité utiliser : « Pour le contrôle fin, je dois utiliser des unités plus précises. »

La valeur de timeout courante est retrouvée à l'aide de `getMaxInactiveInterval()` :

```
public int HttpSession.getMaxInactiveInterval()
```

Cette méthode renvoie la valeur de timeout pour cette session, en secondes. Si vous n'utilisez pas la balise `<session-timeout>`, il est possible d'appeler cette méthode sur une nouvelle session pour déterminer le timeout par défaut de votre serveur.

L'*exemple 7-6* montre l'emploi de ces méthodes dans une servlet qui affiche la valeur de timeout courante, puis qui définit la nouvelle valeur à deux heures. À sa première exécution, le timeout courant affiché correspond à celui de toute l'application. À sa deuxième exécution, le timeout courant affiche deux heures, car c'est la valeur donnée lors de la première exécution.

Exemple 7-6. Jouer avec le timeout

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionTimer extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir l'objet de session courante, en créer un si nécessaire.
    HttpSession session = req.getSession();

    out.println("<HTML><HEAD><TITLE>SessionTimer</TITLE></HEAD>");
    out.println("<BODY><H1>Timeout des sessions</H1>");

    // Afficher le timeout précédent.
    out.println("Le timeout précédent était " +
                session.getMaxInactiveInterval());
    out.println("<BR>");

    // Fixer le nouveau timeout.
    session.setMaxInactiveInterval(2*60*60);  // Deux heures.

    // Afficher le nouveau timeout.
    out.println("Le nouveau timeout est " +
                session.getMaxInactiveInterval());

    out.println("</BODY></HTML>");
  }
}
```

Choisir le bon timeout

Nous disposons donc de plusieurs moyens de contrôle des timeouts de session, mais quelle est la meilleure valeur ? Bien entendu, la réponse est ça dépend.

Il faut d'abord se rappeler que la valeur de timeout de la session ne détermine pas la durée de la session. Elle indique la durée pendant laquelle le serveur attend entre les requêtes avant d'invalider la session. Une session dont le timeout est d'une demi-heure peut sans problème durer plusieurs heures.

Déterminer la bonne durée d'inactivité résulte d'un compromis entre l'aspect pratique pour l'utilisateur, la sécurité de l'utilisateur et la montée en charge du serveur. Les timeouts plus longs apportent plus de souplesse à l'utilisateur car il peut faire des pauses plus longues entre les requêtes, peut-être passer un appel téléphonique ou vérifier s'il a des messages, sans perdre son état. Les timeouts plus courts augmentent sa sécurité car

ils limitent le temps de vulnérabilité si l'utilisateur oublie de fermer sa session et ils augmentent les capacités du serveur car celui-ci libère plus rapidement les objets dans les session.

Comme point de départ, demandez-vous quelle est la durée maximale des pauses de vos utilisateurs entre des requêtes. La réponse type est une demi-heure.

Prenez votre réponse et appliquez quelques règles générales :

- Les applications web demandant une certaine sécurité, comme les banques en ligne, doivent avoir des timeouts plus courts afin de réduire les risques d'utilisateurs d'un navigateur abandonné.
- Les sessions qui stockent des éléments coûteux (comme les connexions à une base de données) doivent aussi avoir des timeouts plus courts afin que le serveur puisse récupérer ou libérer les éléments aussi vite que possible.
- Les sessions qui stockent des éléments non coûteux peuvent avoir des timeouts plus longs.
- Les sessions qui stockent les articles d'un caddie (chers ou non !) doivent avoir des timeouts plus longs car les utilisateurs peuvent oublier une partie du contenu de leur caddie. Une invalidation anticipée vous coûtera alors de l'argent !
- Les sessions mettant dans un cache des informations d'une base de données doivent avoir des timeouts plus courts si le cache est grand, et des timeouts plus longs si la base de données est particulièrement lente.
- Si vous éduquez vos utilisateurs à se fermer leur session quand ils ont terminé, le timeout par défaut peut être allongé.

Enfin, rappelez-vous que le timeout n'est pas obligatoirement le même pour tous les utilisateurs. Vous pouvez utiliser la méthode `setMaxInactiveInterval()` pour définir une timeout personnalisé basé sur les préférences de l'utilisateur ou même modifier le timeout pendant la vie de la session, peut-être pour le rendre plus court après avoir stocké un élément coûteux.

Méthodes du cycle de vie

Plusieurs autres méthodes sont impliquées dans le cycle de vie d'une session :

public boolean HttpSession.isNew()

Cette méthode indique si la session est nouvelle. Une session est considérée nouvelle si elle a été créée par le serveur mais que le client n'a pas encore confirmé qu'il se joignait à la session.

public void HttpSession.invalidate()

Cette méthode provoque l'invalidation immédiate de la session. Tous les objets stockés dans la session sont déliés. Appelez cette méthode pour implémenter une fermeture de session.

public long HttpSession.getCreationTime()

Cette méthode retourne le moment auquel la session a été créée, sous forme d'un long qui représente le nombre de millisecondes depuis l'origine (le 1er janvier 1970, GMT, à minuit).

public long HttpSession.getLastAccessedTime()

Cette méthode retourne le moment auquel le client a envoyé sa dernière requête associée à cette session, sous forme d'un `long` qui représente le nombre de millisecondes depuis l'origine. La requête courante ne représente pas la dernière requête.

Chacune de ces méthodes peut lancer une `java.lang.IllegalStateException` si la session en cours d'accès est invalide.

Invalider manuellement une session inactive

Pour mieux comprendre ces méthodes, l'*exemple 7-7* montre une servlet qui invalide manuellement une session si elle a plus d'un jour ou si elle est inactive depuis plus d'une heure.

Exemple 7-7. Invalider une session inactive

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ManualInvalidate extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir l'objet de session courante, en créer un si nécessaire
    HttpSession session = req.getSession();

    // Invalider la session si elle a plus d'un jour ou si elle a été
    // inactive pendant plus d'une heure.
    if (!session.isNew()) {  // Sauter les nouvelles sessions.
      Date dayAgo = new Date(System.currentTimeMillis() - 24*60*60*1000);
      Date hourAgo = new Date(System.currentTimeMillis() - 60*60*1000);
      Date created = new Date(session.getCreationTime());
      Date accessed = new Date(session.getLastAccessedTime());

      if (created.before(dayAgo) || accessed.before(hourAgo)) {
        session.invalidate();
        session = req.getSession();  // Obtenir une nouvelle session.
      }
    }

    // Continuer le traitement...
  }
}
```

Fonctionnement réel des sessions

Bon, comment un serveur web implémente-t-il le suivi de session ? Lorsqu'un utilisateur accède pour la première fois à une application web, on lui affecte un nouvel objet `HttpSession` et un identifiant de session unique. L'identifiant de session identifie l'utilisateur et il sert à retrouver l'objet `HttpSession` correspondant lors des requêtes suivantes. Certains serveurs utilisent un seul identifiant de session pour l'ensemble du serveur, chaque application web associant cet identifiant à une instance de `HttpSession` différente. D'autres serveurs attribuent un identifiant de session à chaque application web, une précaution supplémentaire pour les applications web malveillante facilitant la falsification d'identité.

En coulisses, l'identifiant de session est habituellement sauvé sur le client dans un cookie appelé `JSESSIONID`. Pour les clients qui ne supportent pas les cookies, l'identifiant de session peut être envoyé dans une URL réécrite, codé dans un paramètre de chemin appelé `jsessionid`, par exemple *http://www.servlets.com/catalogue/servlet/ItemDisplay;jsessionid=123?item=156592391X*. D'autres implémentations, comme l'utilisation des sessions SSL (*Secure Sockets Layer*), sont aussi possibles.

Une servlet détermine l'identifiant de session avec la méthode `getId()` :

```
public String HttpSession.getId()
```

Cette méthode retourne l'identifiant unique associé à cette session dans un `String`. Par exemple, un ID du serveur Tomcat pourrait ressembler à `awj4qyhsn2`. La méthode lance une `IllegalStateException` si la session est invalide.

Souvenez-vous que l'identifiant de session doit être traité comme un secret du serveur. Faites attention à l'emploi de cette valeur.

Obsolescence de HttpSessionContext

Dans l'API Servlet 2.0, il existait un objet `HttpSessionContext` que l'on pouvait utiliser pour examiner les sessions courantes (dont les identifiants de session correspondants) gérées par le serveur. En général cette classe était utilisée pour le débogage et vérifier quelles sessions existaient. Elle existe toujours pour des raisons de compatibilité mais, à partir de l'API Servlet 2.1, son utilisation est désapprouvée et elle est vide. En effet, les identifiants de session doivent être protégés afin qu'ils ne puissent être accédés facilement à partir d'un seul emplacement, en particulier quand l'existence de cet emplacement n'apporte pas un bénéficie suffisant en dehors du débogage.

Suivi de session basé sur une applet

Presque tous les serveurs qui supportent les servlets implémentent le suivi de session à base de cookie, où l'identifiant de session est sauvé sur le client dans un cookie persistant. Le serveur lit l'identifiant de session à partir du cookie `JSESSIONID` et détermine l'objet de session à rendre disponible à chaque requête.

Pour les clients qui sont des applets, cela peut présenter un problème. La plupart des environnements d'applets implémentent `HttpURLConnection` de manière à ce que, au

moment où l'applet effectue une connexion HTTP, l'environnement ajouter automatiquement les cookies du navigateur à la requête. Cela permet à l'applet de participer à la même session que les autres requêtes du navigateur. Le problème est que les autres environnements d'applets, comme les anciennes versions de l'environnement du Java Plug-In, ne disposent pas de cette intégration avec le navigateur et les requêtes de l'applet apparaissent donc séparées de la session normale du navigateur. La solution dans le cas où des applets doivent s'exécuter dans de tels environnements est d'envoyer l'identifiant de session à l'applet et la laisser le retourner au serveur comme un cookie `JSESSIONID` créé artificiellement. Le chapitre 10, *Communication applet-servlet*, présente une classe `HttpMessage` facilitant cette tâche. L'applet peut recevoir l'identifiant de session dans un paramètre d'applet normal (ajouté dynamiquement à la page HTML contenant l'applet).

Quand les cookies ne sont pas supportés

La spécification des servlets imposent aux serveurs web de supporter le suivi de session même pour les navigateurs qui n'acceptent pas les cookies. Beaucoup emploient donc la réécriture d'URL comme solution de repli. Cela demande une aide supplémentaire de la part des servlets qui génèrent des pages contenant des URL. Pour qu'une servlet supporte le suivi de session par réécriture d'URL, elle doit réécrire chaque URL locale avant de l'envoyer au client. L'API Servlet fournit deux méthodes, `encodeURL()` et `encodeRedirectURL()`, pour effectuer ce codage :

`public String HttpServletResponse.encodeURL(String url)`

Cette méthode code (réécrit) l'URL spécifiée pour y inclure l'identifiant de session et retourner la nouvelle URL, ou, si le codage n'est pas nécessaire ou non supporté, elle ne change pas l'URL. Les règles utilisées pour choisir quand et comment coder une URL sont propres au serveur. Toutes les URL émises par une servlet doivent être passées à cette méthode. Notez que cette méthode `encodeURL()` devrait être nommée `rewriteURL()` afin de ne pas la confondre avec le processus de codage d'une URL qui convertit les caractères spéciaux dans une URL.

`public String HttpServletResponse.encodeRedirectURL(String url)`

Cette méthode code (réécrit) l'URL spécifiée afin d'y inclure l'identifiant de session et retourne la nouvelle URL, ou, si le codage n'est pas nécessaire ou non supporté, elle ne change pas l'URL. Les règles utilisées pour choisir quand et comment coder une URL sont propres au serveur. Cette méthode peut utiliser des règles différentes de `encodeUrl()`. Toutes les URL passées à la méthode `sendRedirect()` de `HttpServletResponse` doivent être passées à cette méthode.

Le bout de code suivant montre une servlet écrivant un lien sur elle-même codé de façon à contenir l'identifiant de session courante :

```
out.println("Cliquez <A HREF=\"" +
            res.encodeURL(req.getRequestURI()) + "\">ici</A>");
out.println("pour recharger la page.");
```

Sur les serveurs qui ne supportent pas la réécriture d'URL ou qui l'ont désactivée, l'URL résultante est inchangée. Maintenant, voici un bout de code qui montre une servlet redirigeant l'utilisateur vers une URL codée pour contenir l'identifiant de session :

```
res.sendRedirect(res.encodeRedirectURL("/servlet/NewServlet"));
```

Une servlet peut détecter si l'identifiant de session utilisé pour identifier l'objet HttpSession courant vient d'un cookie ou d'une URL codée en utilisant les méthodes isRequestedSessionIdFromCookie() et isRequestedSessionIdFromUrl() :

```
public boolean HttpServletRequest.isRequestedSessionIdFromCookie()
public boolean HttpServletRequest.isRequestedSessionIdFromURL()
```

Déterminer si l'identifiant de session provient d'une autre source, comme une session SSL, n'est pas encore possible.

Un identifiant de session demandé peut ne pas correspondre à l'identifiant de session retourné par la méthode getSession(), comme lorsque l'identifiant de session est invalide. Une servlet peut déterminer si un identifiant de session demandé est valide en utilisant isRequestedSessionIdValid() :

```
public boolean HttpServletRequest.isRequestedSessionIdValid()
```

Enfin, sachez que lors de l'utilisation du suivi de session basé sur la réécriture d'URL, plusieurs fenêtres du navigateur peuvent appartenir à des sessions différentes ou à la même session, en fonction de la manière dont la fenêtre a été créée et si le lien qui a permis de créer la fenêtre a été passé à la réécriture d'URL.

Examen des sessions

La servlet SessionSnoop montrée à l'*exemple 7-8* utilise la plupart des méthodes présentées jusqu'ici dans ce chapitre afin d'obtenir des informations sur la session courante. La *figure 7-3* montre un exemple de sa sortie.

Exemple 7-8. Examiner les informations d'une session

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionSnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir l'objet de session courante, en créer un si nécessaire.
    HttpSession session = req.getSession();

    // Incrémenter le compteur de hits pour cette page. La valeur est sauvée
    // dans la session du client sous le nom "snoop.count".
    Integer count = (Integer)session.getAttribute("snoop.count");
    if (count == null)
      count = new Integer(1);
    else
      count = new Integer(count.intValue() + 1);
    session.setAttribute("snoop.count", count);
```

Exemple 7-8. Examiner les informations d'une session (suite)

```
    out.println("<HTML><HEAD><TITLE>Examen d'une session</TITLE></HEAD>");
    out.println("<BODY><H1>Examen d'une session</H1>");

    // Afficher le compteur de hits pour cette page.
    out.println("Vous avez visité cette page " + count + "fois.");

    out.println("<P>");

    out.println("<H3>Voici les données de votre session :</H3>");
    Enumeration enum = session.getAttributeNames();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      out.println(name + " : " + session.getAttribute(name) + "<BR>");
    }

    out.println("<H3>Voici des statistiques sur votre session :</H3>");
    out.println("Id de session : " + session.getId() +
                " <I>(à garder secret)</I><BR>");
    out.println("Nouvelle session : " + session.isNew() + "<BR>");
    out.println("Timeout : " + session.getMaxInactiveInterval());
    out.println("<I>(" + session.getMaxInactiveInterval() / 60 +
                " minutes)</I><BR>");
    out.println("Date de création : " + session.getCreationTime());
    out.println("<I>(" + new Date(session.getCreationTime()) + ")</I><BR>");
    out.println("Date de dernier accès : " + session.getLastAccessedTime());
    out.println("<I>(" + new Date(session.getLastAccessedTime()) +
                ")</I><BR>");

    out.println("Id de session provenant d'un cookie : " +
                req.isRequestedSessionIdFromCookie() + "<BR>");
    out.println("Id de session provenant d'une URL : " +
                req.isRequestedSessionIdFromURL() + "<BR>");
    out.println("Id de session valide : " +
                 req.isRequestedSessionIdValid() + "<BR>");

    out.println("<H3>Test de la réécriture d'URL</H3>");
    out.println("Cliquez <A HREF=\"" +
                res.encodeURL(req.getRequestURI()) + "\">ici</A>");
    out.println("pour tester si le suivi de session fonctionne via la");
    out.println("réécriture d'URL même quand les cookies ne sont pas");
    out.println("supportés.");

    out.println("</BODY></HTML>");
  }
}
```

Cette servlet commence par le même code que la servlet SessionTracker montrée à l'*exemple 7-4*. Ensuite elle poursuit en affichant l'identifiant de session courante, si c'est une nouvelle session, la valeur de son timeout, la date de création de la session et la date

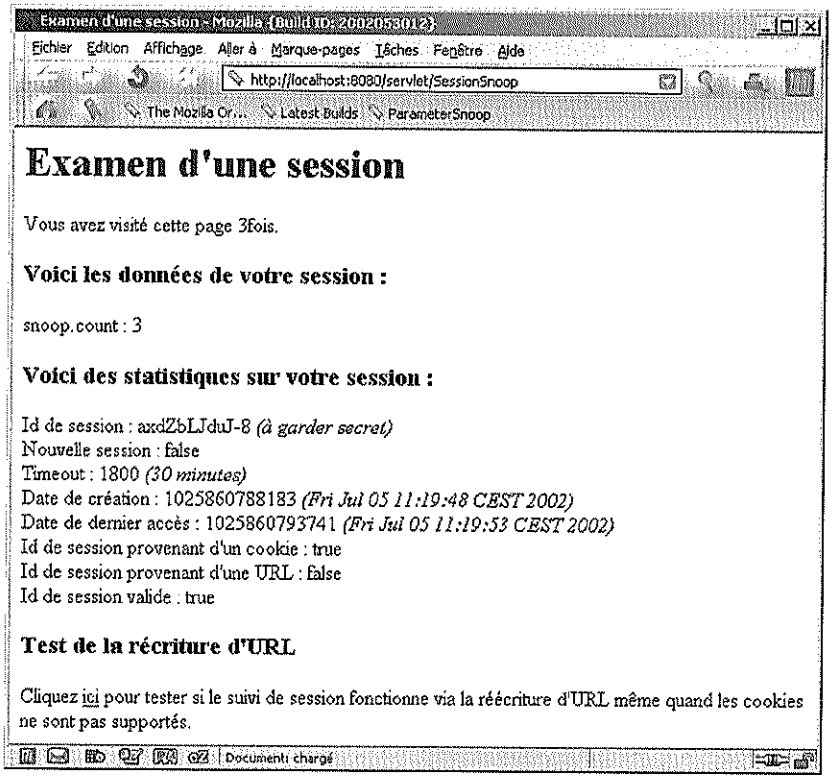


Figure 7-3. Exemple de sortie de SessionSnoop

du dernier accès de la session. Ensuite, la servlet affiche si l'identifiant de session demandé (s'il y en a un) provient d'un cookie ou d'une URL et s'il est valide. Enfin, la servlet montre une URL codée qui peut être utilisée pour recharger cette page afin de tester si la réécriture d'URL fonctionne même lorsque les cookies ne sont pas supportés.

Événement de liaison de session

Certains objets pourraient souhaiter effectuer une action lorsqu'ils sont liés ou déliés d'une session. Par exemple, une connexion à une base de données pourrait commencer une transaction lorsqu'elle est liée à une session et la terminer lorsqu'elle est déliée. Tout objet implémentant l'interface `javax.servlet.http.HttpSessionBindingListener` est averti lorsqu'il est lié ou délié d'une session. L'interface déclare deux méthodes, `valueBound()` et `valueUnbound()`, qui doivent être implémentées :

```
public void HttpSessionBindingListener.valueBound(
            HttpSessionBindingEvent event)
public void HttpSessionBindingListener.valueUnbound(
            HttpSessionBindingEvent event)
```

La méthode `valueBound()` est appelée lorsque celui qui écoute est lié à une session, et `valueUnbound()` est appelée lorsqu'il en est délié, soit parce qu'il est retiré ou remplacé ou parce que la session devient invalide.

L'argument javax.servlet.http.HttpSessionBindingEvent fournit l'accès au nom sous lequel l'objet va être lié (ou délié) *via* la méthode getName() :

```
public String HttpSessionBindingEvent.getName()
```

L'objet HttpSessionBindingEvent fournit aussi l'accès à l'objet HttpSession auquel celui qui écoute est lié (ou délié) *via* la méthode getSession() :

```
public HttpSession HttpSessionBindingEvent.getSession()
```

L'*exemple 7-8* montre l'utilisation de HttpSessionBindingListener et HttpSessionBindingEvent par un objet qui journalise les événements de liaison à une session.

Exemple 7-9. Tracer les événements de liaison d'une session

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionBindings extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    // Obtenir l'objet de session courante, en créer un si nécessaire.
    HttpSession session = req.getSession();

    // Ajouter un CustomBindingListener.
    session.setAttribute("bindings.listener",
                         new CustomBindingListener(getServletContext()));

    out.println("Cette page est intentionnellement vide");
  }
}

class CustomBindingListener implements HttpSessionBindingListener {

  // Sauver un ServletContext à utiliser pour sa méthode log().
  ServletContext context;

  public CustomBindingListener(ServletContext context) {
    this.context = context;
  }

  public void valueBound(HttpSessionBindingEvent event) {
    context.log("[" + new Date() + "] LIÉ comme " + event.getName() +
                " à " + event.getSession().getId());
  }

  public void valueUnbound(HttpSessionBindingEvent event) {
```

Exemple 7-9. Tracer les événements de liaison d'une session (suite)

```
    context.log("[" + new Date() + "] DÉLIÉ comme " + event.getName() +
                " de " + event.getSession().getId());
  }
}
```

Chaque fois qu'un objet `CustomBindingListener` est lié à une session, sa méthode `valueBound()` est appelée et l'événement est journalisé. Chaque fois qu'il est délié d'une session, sa méthode `valueUnbound()` est appelée de façon à journaliser cet événement. Nous pouvons observer la séquence des événements en examinant le journal des événements du serveur.

Supposons que cette servlet est appelée une fois, rechargée 30 secondes plus tard, et n'est pas appelée avant au moins une demi-heure. Le journal d'événements devrait ressembler à cela :

```
[Tue Sep 27 22:46:48 PST 2000]
  LIÉ comme bindings.listener à awj4qyhsn2
[Tue Sep 27 22:47:18 PST 2000]
  DÉLIÉ comme bindings.listener de awj4qyhsn2
[Tue Sep 27 22:47:18 PST 2000]
  LIÉ comme bindings.listener à awj4qyhsn2
[Tue Sep 27 23:17:18 PST 2000]
  DÉLIÉ comme bindings.listener de awj4qyhsn2
```

La première entrée survient lors de la première requête de la page, lorsque celui qui écoute est lié à la nouvelle session. La deuxième et la troisième entrée correspondent au rechargement, quand l'auditeur est délié puis relié pendant le même appel à `setAttribute()`. La quatrième entrée arrive une demi-heure plus tard, lorsque la session expire et qu'elle est invalidée.

Faire ses courses en utilisant le suivi de session

Terminons ce chapitre par la remarquable simplification de notre servlet de visualisation d'un caddie lorsque nous utilisons le suivi de session. L'*exemple 7-9* montre le visualiseur sauvant chaque article du caddie de la session de l'utilisateur sous le nom `cart.items`. Notez que les URL de la page sont réécrites afin de supporter les clients qui ont désactivé les cookies.

Exemple 7-10. Utiliser l'API de suivi de session

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerSession extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
```

Exemple 7-10. Utiliser l'API de suivi de session (suite)

```
    // Obtenir l'objet de session courante, en créer un si nécessaire.
    HttpSession session = req.getSession();

    // Les articles du caddie sont maintenus dans l'objet de session.
    String[] items = (String[])session.getAttribute("cart.items");

    out.println("<HTML><HEAD><TITLE>Suivi de session</TITLE></HEAD>");
    out.println("<BODY><H1>Démonstration du suivi de session</H1>");

    // Afficher le contenu du caddie.
    out.println("Voici les articles qui se trouvent dans votre caddie :<BR>");
    if (items == null) {
      out.println("<B>Aucun</B>");
    }
    else {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++) {
        out.println("<LI>" + items[i]);
      }
      out.println("</UL>");
    }

    // Demander si l'utilisateur veut ajouter d'autres articles ou payer.
    out.println("<FORM ACTION=\"" +
                res.encodeURL("/servlet/ShoppingCart") + "\" METHOD=POST>");
    out.println("Voulez-vous ?<BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Ajouter d'autres articles \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Passer au paiement \">");
    out.println("</FORM>");

    // Proposer une page d'aide. La coder si nécessaire.
    out.println("Pour de l'aide, cliquez <A HREF=\"" +
                res.encodeURL("/servlet/Help?topic=ShoppingCartViewer") +
                "\">ici</A>");

    out.println("</BODY></HTML>");
  }
}
```

8

Sécurité

Jusqu'à présent, nous avons imaginé que nos servlets existent dans un monde parfait, où l'on peut faire confiance à tout le monde et où personne ne ferme sa porte à clé le soir. Malheureusement, c'est un monde fantastique des années 50 : la vérité est que l'Internet a sa part de mauvais plaisants. Depuis que les entreprises mettent de plus en plus d'importance sur le commerce en ligne et commencent à construire des intranets avec des informations sensibles, la sécurité est devenue un des sujets les plus importants dans la programmation web.

La sécurité consiste à conserver les informations sensibles dans les mains des utilisateurs autorisés. Sur le Web, cela se résume à quatre aspects importants :

Authentification

Être capable de vérifier l'identité des parties impliquées.

Habilitation

Limiter l'accès aux ressources à un ensemble d'utilisateurs ou de programmes sélectionnés.

Confidentialité

Garantir que seules les parties impliquées comprennent la communication.

Intégrité

Être capable de vérifier que le contenu de la communication n'est pas changé pendant la transmission.

Il est plus facile de comprendre ces aspects en se plaçant dans le contexte suivant : un client veut être sûr qu'il parle à un serveur légitime (authentification) et il veut aussi être sûr que toute information qu'il transmet, comme son numéro de carte de crédit, n'est pas sujette aux indiscrétions (confidentialité). Le serveur est aussi concerné par l'authentification et la confidentialité. Si une entreprise vend un service ou fournit des informations sensibles à ses propres employés, elle a vraiment intérêt à s'assurer que personne, en dehors des utilisateurs autorisés, n'y a accès. Et les deux côtés ont besoin de l'intégrité afin d'être sûrs que toute information qu'ils s'envoient arrive non modifiée.

Authentification, habilitation, confidentialité et intégrité sont liées par la technologie des certificats numériques. Les certificats numériques permettent aux serveurs web et aux clients d'utiliser des techniques de cryptographie avancées pour gérer l'identification et le chiffrement de façon sûre. Grâce au support des certificats numériques intégré à Java, les servlets sont une plate-forme idéale pour déployer des applications web sûres utilisant cette technologie. Nous y reviendrons en détail plus loin.

La sécurité implique aussi de garantir que les pirates ne puissent avoir accès aux données sensibles de votre serveur web. Java ayant été conçu dès le départ comme un langage orienté réseau sûr, il est possible d'exploiter les fonctions de sécurité intégrées et de garantir que les extensions du serveur provenant d'un fournisseur tiers sont au moins aussi sûres que celles que vous avez écrites.

Ce chapitre introduit les bases de la sécurité web et la technologie des certificats numériques dans le contexte de l'utilisation des servlets. Il explique aussi comment maintenir la sécurité de votre serveur web lorsque des servlets de fournisseurs tiers non sûrs s'exécutent. Vous noterez que ce chapitre prend une approche de haut niveau et montre moins d'exemples que les précédents. La raison en est que plusieurs aspects de ce chapitre demandent de mettre en œuvre une administration spécifique au serveur web. Les servlets ne sont là que pour l'accompagnement.

Enfin, un avertissement. Nous ne sommes que deux programmeurs de servlets et nous rejetons toute responsabilité pour tout incident lié à la sécurité qui pourrait résulter de nos conseils. Pour plus de détails sur les technologies et les procédures de sécurité sur le Web, lisez le livre *Web Security & Commerce* de Simson Garfinkel avec la participation de Gene Spafford (O'Reilly & Associates). Bien sûr, eux aussi rejetteront probablement toute responsabilité.

Authentification HTTP

Comme nous en avons parlé rapidement au chapitre 4, *Retrouver l'information*, le protocole HTTP intègre un support de l'authentification, appelé *authentification de base*, basé sur un modèle simple de demande/réponse, nom d'utilisateur/mot de passe. Avec cette technique, le serveur web maintient une base de données de noms d'utilisateurs et de mots de passe, et il identifie certaines ressources (fichiers, répertoires, servlets, etc.) comme étant protégées. Lorsqu'un utilisateur demande l'accès à une ressource protégée, le serveur répond en lui demandant son nom et son mot de passe. À ce moment-là, le navigateur affiche généralement une boîte de dialogue dans laquelle l'utilisateur entre les informations demandées, qui sont ensuite retournées au serveur dans une seconde requête autorisée. Si le nom et le mot de passe soumis correspondent aux informations dans la base de données du serveur, l'accès est autorisé. Tout le processus d'authentification est pris en charge par le serveur lui-même.

L'authentification de base est très faible. Elle ne fournit ni confidentialité ni intégrité, et l'authentification est des plus basique. Le problème est que les mots de passe sont transmis sur le réseau, légèrement déguisés par un codage Base64 bien connu et facilement réversible. Toute personne analysant le flot de données TCP/IP a un accès complet et immédiat à toutes les informations échangées, y compris le nom d'utilisateur et le mot de passe, sauf si un chiffrement SSL est utilisé (comme présenté plus loin). De plus, les mots de passe sont souvent stockés en clair sur le serveur, les rendant vulnérables à

toute personne piratant le système de fichiers. Bien que ce soit mieux que rien, les sites basés uniquement sur l'authentification de base ne peuvent pas être considérés comme vraiment sûrs.

L'*authentification digest* est une variation du schéma de l'authentification de base. Au lieu de transmettre directement un mot de passe sur le réseau, un digest du mot de passe est utilisé. Ce digest est produit en prenant une clé de hachage (en utilisant un algorithme de cryptage MD5 très sûr) du nom de l'utilisateur, du mot de passe, de l'URI, de la méthode de la requête HTTP et d'une valeur *occasionnelle* (*nonce*) générée aléatoirement par le serveur. Si les digests correspondent, l'accès est autorisé. Les transactions sont ainsi un peu plus sûres car les digests ne sont valides que pour une seule requête d'URI et une valeur occasionnelle. Cependant, le serveur doit toujours maintenir une base de données des mots de passe originels. Et, au moment de l'écriture de ce livre, l'authentification digest n'est pas supportée par beaucoup de navigateurs.

La morale de cette histoire est que l'authentification HTTP peut être utile dans des environnements de faible sécurité. Par exemple, un site qui fait payer l'accès à son contenu, disons un journal en ligne, est plus concerné par la facilité d'utilisation et d'administration que par une sécurité forte. L'authentification HTTP est alors souvent suffisante.

Configurer l'authentification HTTP

Dans les versions de l'API Servlet antérieures à 2.2, la technique de configuration varie selon le serveur. À partir de l'API 2.2, elle a été standardisée et la configuration des politiques de sécurité peut maintenant être accomplie de manière portable à l'aide du descripteur de déploiement *web.xml*.

Avant de commencer, faisons deux mises en garde à propos de cette portabilité. Tout d'abord, un serveur web n'est pas obligé d'implémenter le mécanisme de sécurité de l'API Servlet 2.2 pour être considéré comme compatible avec cette API. L'implémentation complète de la sécurité est plutôt complexe et un conteneur de servlets est autorisé à n'en implémenter qu'une partie ou même pas du tout. Les seuls serveurs qui ont l'obligation d'implémenter le mécanisme de sécurité complet sont ceux qui veulent être compatibles avec la spécification de Java 2, Enterprise Edition (J2EE), de laquelle l'API Servlet fait partie. Ensuite, la sécurité est l'un des aspects les plus récents et les moins compris de l'API Servlet 2.2. L'implémentation de ces mécanismes dans les serveurs web a quelque peu évolué. Tout cela finira par se mettre en place, mais pour le moment, afin de garantir que votre site reste sûr, assurez-vous de tester tout au moins une fois lorsque vous migrez une application web sur un autre serveur.

Authentification basée sur les rôles

Avec des balises du descripteur de déploiement d'une application web, il est possible de définir des contraintes de sécurité indiquant que certaines pages d'une application web ne peuvent être accédées que par les utilisateurs ayant les bons certificats. Les servlets emploient l'habilitation basée sur les rôles pour gérer les accès. Dans ce modèle, les permissions d'accès sont accordées à une entité abstraite appelée un *rôle de sécurité*, et l'accès n'est accordé qu'aux utilisateurs ou aux groupes d'utilisateurs faisant partie d'un rôle donné. Par exemple, vous pouvez configurer votre site afin que les pages affichant les salaires ne puissent être consultées que par les personnes du rôle « directeur ».

Le descripteur de déploiement spécifie le type d'accès accordé à chaque rôle, mais il ne spécifie pas la liaison de ce rôle à un utilisateur ou à un groupe. Cela se fait pendant le déploiement de l'application en utilisant des outils spécifiques au serveur. La correspondance peut provenir de plusieurs sources (fichiers texte, base de données, système d'exploitation, etc.).

Restreindre l'accès à une servlet

Pour prendre un exemple concret, supposons que nous ayons une servlet dont l'accès doit être restreint, comme montré à l'*exemple 8-1*. (Les servlets ne sont pas les seules à pouvoir être protégées. C'est aussi le cas des fichiers statiques et de tout le reste. Mais, d'une certaine manière, il semble approprié à ce livre que nous utilisions une servlet comme exemple.)

Exemple 8-1. Êtes-vous sûr d'avoir le droit le lire cet exemple ?

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SalaryServer extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    out.println("Information top-secrète :");
    out.println("Tous les autres sont mieux payés que vous !");
  }
}
```

Supposons aussi que le serveur offre une base de données des utilisateurs contenant une liste de noms d'utilisateurs, de mots de passe et de rôles. Souvenez-vous que tous les serveurs n'offrent pas cette possibilité et que ceux qui le font ont la possibilité d'implémenter la base de données des utilisateurs comme ils le souhaitent. Pour Tomcat 3.2, les utilisateurs se trouvent dans le fichier *conf/tomcat-users.xml* (voir l'*exemple 8-2*). (Les futures versions de Tomcat auront très certainement un format de stockage plus sûr que du texte en clair dans un fichier XML.)

Exemple 8-2. Fichier conf/tomcat-users.xml de Tomcat

```
<tomcat-users>
  <user name="Dilbert"      password="dnrc"
        roles="ingénieur" />
  <user name="Wally"        password="iluvalice"
        roles="ingénieur,paresseux"/>
  <user name="LeBoss" password="LeBoss"
        roles="directeur,paresseux"/>
</tomcat-users>
```

Notez que LeBoss est le seul utilisateur qui appartient au rôle directeur. En supposant qu'il sache comment connecter son ordinateur « télécran » sur le web, il doit être le seul employé à avoir accès à notre servlet secrète. Cette règle est indiquée dans le fichier *web.xml* montré à l'*exemple 8-3*.

Souvenez-vous que l'ordre des balises dans un fichier *web.xml* est important. Suivez toujours l'ordre suivant : <security-constraint>, <login-config>, puis <security-role>. L'ordre des balises dans ces éléments est aussi important.

Cet exemple peut paraître un peu complexe. La sécurité est un sujet pour lequel il est souvent plus facile d'employer un outil graphique d'aide à la création des fichiers *web.xml*.

Exemple 8-3. Restreindre l'accès par une authentification de base

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            secret
        </servlet-name>
        <servlet-class>
            SalaryServer
        </servlet-class>
    </servlet>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>
                SecretProtection
            </web-resource-name>
            <url-pattern>
                /servlet/SalaryServer
            </url-pattern>
            <url-pattern>
                /servlet/secret
            </url-pattern>
            <http-method>
                GET
            </http-method>
            <http-method>
                POST
            </http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>
                directeur
```

Exemple 8-3. Restreindre l'accès par une authentification de base (suite)

```
            </role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>
            BASIC       <!-- BASIC, DIGEST, FORM, CLIENT-CERT -->
        </auth-method>
        <realm-name>
            Default     <!-- optionnel, utile uniquement avec BASIC -->
        </realm-name>
    </login-config>

    <security-role>
        <role-name>
            directeur
        </role-name>
    </security-role>
</web-app>
```

Ce descripteur de déploiement protège tous les accès GET et POST à */servlet/secret* et à */servlet/SalaryServer* afin que seuls les utilisateurs du rôle `directeur` qui se sont connectés à l'aide d'une authentification de base aient un accès autorisé. L'accès au reste du site n'est pas restreint.

Voici comment cela fonctionne. La balise `<security-constraint>` protège une `<web-resource-collection>` afin que l'accès ne soit accordé qu'aux rôles indiqués dans `<auth-constraint>`. Chaque `<web-resource-collection>` contient un nom, un nombre quelconque de motifs d'URL indiquant les URL à protéger et un nombre quelconque de méthodes HTTP pour lesquelles l'accès doit être restreint. Le nom doit être indiqué même s'il n'est réellement utile qu'aux outils. Pour les motifs d'URL, vous pouvez employer les mêmes caractères génériques que ceux de la correspondance de servlet (voir le chapitre 2, *Introduction aux servlets HTTP*). Pour les méthodes HTTP, vous devez généralement indiquer au moins GET et POST ; si aucune entrée `<http-method>` n'est donnée, toutes les méthodes sont protégées. La balise `<auth-constraint>` contient un nombre quelconque de rôles pour lesquels l'accès à l'ensemble des ressources doit être accordé.

La balise `<login-config>` spécifie la méthodologie d'ouverture de session employée par l'application. Dans le cas présent, nous indiquons une authentification de base dans le domaine `Default`. La balise `<auth-method>` accepte les valeurs `BASIC`, `DIGEST`, `FORM` et `CLIENT-CERT` qui représentent respectivement les authentifications de base, digest, basées sur un formulaire et les certificats côté client. Nous reviendrons plus loin sur l'authentification basée sur un formulaire et les certificats numériques. La balise `<realm-name>` spécifie le domaine de connexion à utiliser (puisqu'un serveur peut gérer différentes listes d'utilisateurs pour des domaines différents) ; elle n'a un sens que pour les authentifications de base et digest.

Enfin, la balise `<security-role>` contient une liste de rôles pouvant être utilisés dans cette application. La déclaration explicite des rôles à employer permet une manipulation du fichier à l'aide d'outils.

Les règles de sécurité sont plutôt limitées. Par exemple, il n'est pas possible d'autoriser les accès à tous les utilisateurs excepté ceux du rôle « liste noire ». La solution consiste à mettre en place un rôle de liste approuvée que le serveur associe à tous les utilisateurs qui ne sont pas dans certains groupes.

Maintenant que ce fichier *web.xml* est en place, le serveur intercepte toutes les requêtes conformes aux motifs d'URL donnés utilisant les méthodes HTTP indiquées et vérifie les certificats du client. S'ils sont valides et si l'utilisateur se trouve (de manière spécifique au serveur) dans le rôle `directeur`, alors l'accès est accordé. Si les certificats ne sont pas valides ou si l'utilisateur ne fait pas partie du rôle `directeur`, alors l'accès est refusé et le navigateur affiche une fenêtre demandant à l'utilisateur de recommencer.

Retrouver l'information d'authentification

Une servlet peut retrouver les informations sur l'authentification du serveur en utilisant deux méthodes introduites au chapitre 4 : `getRemoteUser()` et `getAuthType()`. L'API Servlet 2.2 inclut aussi une nouvelle méthode, `getUserPrincipal()`, qui renvoie un objet implémentant l'interface `java.security.Principal` :

```
public java.security.Principal HttpServletRequest.getUserPrincipal()
```

Un *principal* est le terme technique désignant l'entité en train d'être authentifiée. Il peut s'agir d'un groupe, d'une société ou juste un identifiant de session générique. L'interface `Principal` inclut une méthode `getName()` qui retourne le nom du principal. Pour déterminer l'identité de l'utilisateur authentifié, le meilleur moyen est d'employer la méthode `getUserPrincipal()`, `getRemoteUser()` étant principalement là pour une compatibilité avec CGI. La méthode `isUserInRole()` a aussi été introduite dans l'API Servlet 2.2. Elle retourne `true` uniquement si l'utilisateur authentifié appartient au rôle indiqué :

```
public boolean HttpServletRequest.isUserInRole(String role)
```

Cette méthode permet de place une certaine prise de décision dans la servlet elle-même. Par exemple, supposons qu'un descripteur de déploiement autorise les accès à une servlet par plusieurs rôles différents. En appelant cette méthode, la servlet peut varier son comportement à l'exécution en fonction du rôle de l'utilisateur authentifié.

Il est possible de créer dans le descripteur de déploiement autant d'alias de rôles que souhaité afin que, du point de vue de la servlet, une requête sur le rôle `dir` soit identique à une requête sur le rôle `directeur`. Cela s'avère particulièrement utile lors de l'intégration de servlets d'applications web pouvant employer des noms de rôle différents de ceux de l'application web courante. Les alias sont configurés pour chaque servlet à l'aide de la balise `<security-role-ref>` dans la balise `<servlet>`, comme le montre l'extrait de *web.xml* suivant :

```
<servlet>
    <servlet-name>
        secret
    </servlet-name>
    <servlet-class>
        SalaryViewer
    </servlet-class>
    <security-role-ref>
```

```
        <role-name>
            dir        <!-- nom utilisé par la servlet -->
        </role-name>
        <role-link>
            directeur  <!-- nom dans le descripteur de déploiement -->
        </role-link>
    </security-role-ref>
</servlet>
```

Chaque servlet peut avoir un nombre quelconque de balises `<security-role-ref>`. Souvenez-vous que ces alias ne sont valides que lors des accès à la servlet sous son nom enregistré.

L'*exemple 8-4* montre une servlet simple qui indique au client son nom, son principal, le type d'authentification effectuée (`BASIC`, `DIGEST`, `FORM`, `CLIENT-CERT`) et si l'utilisateur est un `directeur`. Pour voir cette servlet en action, vous devez l'installer sur votre serveur web et la protéger par un schéma de sécurité comme nous l'avons vu à la section précédente (en vous assurant de restreindre les accès à cette nouvelle `<url-pattern>`).

Exemple 8-4. Analyser les informations d'authentification

```
import java.io.*;
import java.security.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AuthenticationSnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HTML><HEAD><TITLE>AuthenticationSnoop</TITLE></HEAD><BODY>");

    out.println("<H1>Voici une ressource protégée par mot de passe</H1>");
    out.println("<PRE>");
    out.println("Nom d'utilisateur : " + req.getRemoteUser());
    String name = (req.getUserPrincipal() == null) ?
                  null : req.getUserPrincipal().getName();
    out.println("Nom de principal : " + name);
    out.println("Type d'authentification : " + req.getAuthType());
    out.println("Est un directeur : " + req.isUserInRole("directeur"));
    out.println("</PRE>");
    out.println("</BODY></HTML>");
  }
}
```

La sortie doit ressembler à la suivante :

```
Voici une ressource protégée par mot de passe
Nom d'utilisateur : jhunter
```

```
Nom de principal : jhunter
Type d'authentification : BASIC
Est un directeur : false
```

Authentification à base de formulaire

Les servlets peuvent aussi effectuer l'authentification sans reposer sur l'authentification HTTP, en utilisant les formulaires HTML. Avec cette technique, les utilisateurs peuvent entrer sur votre site *via* une page de connexion agréable, bien conçue et descriptive. Par exemple, imaginez que vous développiez un site bancaire en ligne. Laisseriez-vous le navigateur présenter une invite générique pour la saisie du nom et du mot de passe, ou offririez-vous à vos clients un formulaire de connexion personnalisé qui demanderait poliment des certificats spécifiques, comme montré à la *figure 8-1* ?

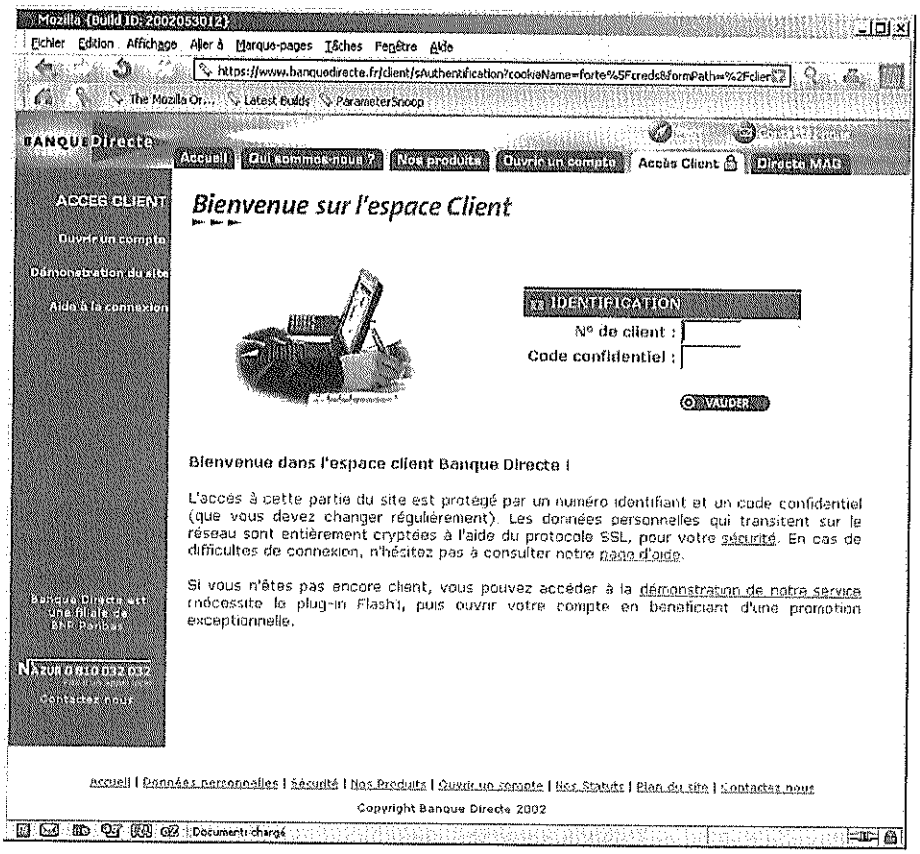


Figure 8-1. Un écran d'ouverture de session pour une banque en ligne

Beaucoup de banques et d'autres services en ligne ont choisi d'utiliser l'authentification à base de formulaire. L'implémentation d'un tel système est relativement simple avec les servlets car elle repose sur l'API Servlet 2.2. Pour passer de l'authentification de base à l'authentification par formulaire, remplacez la section `<login-config>` du fichier *web.xml* de l'*exemple 8-3* par la section `<login-config>` de l'*exemple 8-5*.

Exemple 8-5. Configurer une authentification basée sur un fomulaire

```
<login-config>
    <auth-method>
        FORM            <!-- BASIC, DIGEST, FORM, CLIENT-CERT -->
    </auth-method>
    <form-login-config>  <!-- Utile uniquement avec FORM -->
        <form-login-page>
            /loginpage.html
        </form-login-page>
        <form-error-page>
            /errorpage.html
        </form-error-page>
    </form-login-config>
</login-config>
```

Notez que `<auth-method>` est passé de `BASIC` à `FORM`. Cela indique que l'authentification à base de formulaire doit être utilisée pour cette application web. La balise `<realm-name>` a aussi été remplacée par `<form-login-config>`. Cette balise spécifie la page d'ouverture de session et la page d'erreur à utiliser pour l'authentification. La page d'ouverture de session doit être une page bien conçue, descriptive et agréable demandant les certificats de l'utilisateur. La page d'erreur doit être une page bien conçue, descriptive et indiquant, si possible, au serveur que les certificats ne sont pas valides. Les deux URL doivent être des chemins absolus dont la racine est celle du contexte.

Quand le serveur reçoit une requête pour une ressource protégée, il vérifie si l'utilisateur a déjà ouvert sa session. Par exemple, un serveur peut rechercher un objet `Principal` stocké dans l'objet `HttpSession` de l'utilisateur. S'il le trouve, les rôles du `Principal` sont comparés à ceux requis pour l'accès à la ressource. S'il ne le trouve pas ou si le `Principal` n'appartient pas à un des rôles autorisés, le client est redirigé vers la page d'ouverture de session (mais le serveur commence par enregistrer, probablement dans l'objet `HttpSession` de l'utilisateur, l'URL initialement demandée).

La page d'ouverture de session contient un formulaire dans lequel l'utilisateur peut entrer son nom et son mot de passe et les renvoyer au serveur. L'accès à la ressource est autorisé uniquement si le nom d'utilisateur et le mot de passe sont valides et s'ils forment un `Principal` d'un rôle autorisé, auquel cas le serveur redirige poliment l'utilisateur vers cette ressource. Sinon, il le redirige vers la page d'erreur.

La page d'ouverture de session contient un formulaire incluant des valeurs spéciales afin de garantir que les bonnes données sont envoyées de la bonne manière au serveur. Le formulaire doit employer la méthode POST pour l'URL `j_security_check` (sans barre oblique de tête, même si certains serveurs l'impose de façon erronée) avec le nom d'utilisateur envoyé dans `j_username` et le mot de passe dans `j_password`. Par exemple :

```
<FORM METHOD=POST ACTION="j_security_check">
Nom d'utilisateur : <INPUT TYPE=TEXT NAME="j_username"><br>
Mot de passe : <INPUT TYPE=PASSWORD NAME="j_password"><br>
<INPUT TYPE=SUBMIT>
</FORM>
```

L'*exemple 8-6* montre un fichier *loginpage.html* plus réaliste qui génère le formulaire montré à la *figure 8-2*.

Exemple 8-6. Le fichier loginpage.html

```
<HTML>
<TITLE>Ouverture de session</TITLE>
<BODY>
<FORM METHOD=POST ACTION=j_security_check>
<CENTER>
<TABLE BORDER=0>
<TR><TD COLSPAN=2>
<P ALIGN=center>
Bonjour !  Veuillez entrer votre nom<br>
et votre mot de passe pour vous connecter.
</TD></TR>

<TR><TD>
<P ALIGN=right><B>Nom :</B>
</TD>
<TD>
<P><INPUT TYPE=TEXT NAME="j_username" VALUE="" SIZE=15>
</TD></TR>

<TR><TD>
<P ALIGN=RIGHT><B>Mot de passe :</B>
</TD>
<TD>
<P><INPUT TYPE=PASSWORD NAME="j_password" VALUE="" SIZE=15>
</TD></TR>

<TR><TD COLSPAN=2>
<CENTER>
<INPUT TYPE=submit VALUE="  OK   ">
</CENTER>
</TD></TR>
</TABLE>
</FORM>
</BODY></HTML>
```

La *figure 8-2* montre le formulaire généré.

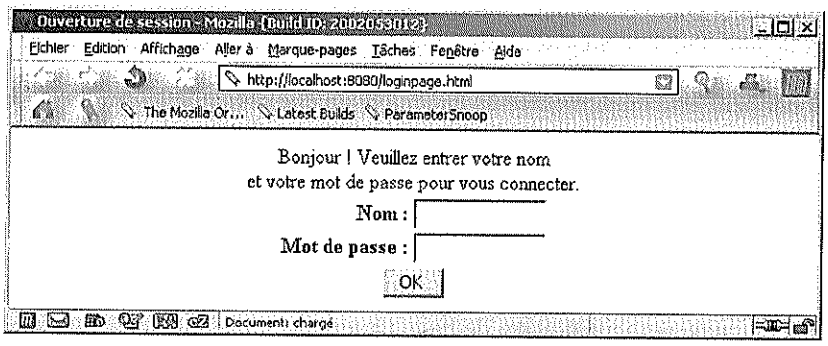


Figure 8-2. Un formulaire d'ouverture de session agréable

La page d'erreur spécifiée dans la section `<login-config>` de *web.xml* peut être n'importe quel fichier HTML. Il n'y a aucune balise spéciale à mettre, et elle n'a malheureusement accès à aucune information spéciale indiquant pourquoi l'accès a été refusé ou même vers quelle page l'utilisateur doit aller pour recommencer. L'*exemple 8-7* donne une page d'erreur simple.

Exemple 8-7. Le fichier errorpage.html

```
<HTML>
<TITLE>Accès interdit</TITLE>
<BODY>
Désolé, votre ouverture de session a été réfusée.
Cliquez sur le bouton Précédent pour recommencer.
</BODY></HTML>
```

Comparé à l'authentification de base, l'ouverture de session par formulaire présente l'avantage de donner à l'utilisateur une page agréable et descriptive. Elle a le même problème que l'authentification de base concernant la transmission en clair du mot de passe sauf si le canal de communication a été sécurisé d'une autre manière.

L'ouverture de session de base et par formulaire ont toutes deux le problème de ne pas offrir un mécanisme de fermeture de session. L'appel à `session.invalidate()` a toutes les chances d'avoir cet effet pour l'ouverture de session basée sur un formulaire, mais ce n'est pas garanti. Toutes deux reposent sur le serveur pour la validation des utilisateurs, même s'il existe des cas pour lesquels la validation doit se faire d'une manière non supportée par le serveur (par exemple, certaines banques demandent un numéro de compte, un mot de passe et un numéro personnel d'identification). Pour résoudre ces problèmes, nous pouvons employer l'authentification personnalisée.

Authentification personnalisée

Normalement, l'authentification du client est gérée par le serveur web. Le descripteur de déploiement indique au serveur quelles ressources doivent être restreintes à quels rôles et le serveur gère d'une façon ou d'une autre la correspondance entre les utilisateurs/groupes et les rôles.

C'est souvent suffisant. Mais parfois la politique de sécurité souhaitée ne peut être implémentée par le serveur. Il est possible que la liste des utilisateurs doive être stockée dans un format non reconnu par le serveur. Ou peut-être que vous souhaitez que n'importe quel nom d'utilisateur soit accepté, tant qu'il est donné avec un mot de passe conforme à un motif de clé. Pour gérer ces situations, nous pouvons utiliser des servlets. Une servlet peut être implémentée de façon à ce qu'elle obtienne les informations sur les utilisateurs à partir d'une base de données relationnelle ou d'un fichier au format spécial ; elle peut aussi être écrite pour garantir la politique de sécurité que vous voulez. Une telle servlet peut même ajouter, supprimer ou manipuler les entrées d'utilisateur, quelque chose qui n'est pas directement supporté dans l'API Servlet, excepté au travers d'extensions propriétaires du serveur

Une servlet utilise les codes d'état et les en-têtes HTTP pour gérer sa propre politique de sécurité de type authentification de base. La servlet reçoit les certificats de l'utilisa-

teur codés dans l'en-tête `Authorization`. Si elle choisit de refuser ces certificats, elle le fait *via* le code d'état `SC_UNAUTHORIZED` et un en-tête `WWW-Authenticate` qui décrit les certificats attendus. Un serveur web gère normalement ces détails sans impliquer ses servlets, mais pour qu'une servlet effectue sa propre habilitation, elle doit les gérer elle-même tandis que le serveur est configuré pour ne pas restreindre l'accès à la servlet.

L'en-tête `Authorization`, s'il est envoyé par le client, contient le nom d'utilisateur et le mot de passe du client. Avec le schéma d'habilitation de base, l'en-tête `Authorization` contient la chaîne *`nom-utilisateur:mot-de-passe`* codée en Base64. Par exemple, le nom `webmaster` associé au mot de passe `try2gueSS` est envoyé dans l'en-tête `Authorization` avec la valeur :

```
Authorization: BASIC d2VibWFzdGVyOnRyeTJndWVTUw
```

Si une servlet en a besoin, elle peut envoyer un en-tête `WWW-Authenticate` pour indiquer au client le schéma d'habilitation et le domaine dans lequel les utilisateurs seront contrôlés. Un *domaine* est simplement un ensemble de comptes utilisateur et de ressources protégées. Par exemple, pour indiquer au client d'utiliser l'authentification de base pour le domaine `Admin`, l'en-tête `WWW-Authenticate` est :

```
WWW-Authenticate: BASIC realm="Admin"
```

L'*exemple 8-8* montre une applet qui effectue une habilitation personnalisée, recevant un en-tête `Authorization` et retournant un code d'état `SC_UNAUTHORIZED` et un en-tête `WWW-Authenticate` si nécessaire. La servlet restreint l'accès à son information top secrète aux utilisateurs (et mots de passe) qu'elle reconnaît dans la liste d'utilisateurs. Pour cet exemple, la liste est gardée dans un `Hashtable` simple et son contenu est figé dans le code ; bien sûr, pour une servlet en exploitation, cela doit être remplacé par un autre mécanisme comme une base de données relationnelle externe.

Pour retrouver le nom d'utilisateur et le mot de passe codés en Base64, la servlet a besoin d'un décodeur Base64. Heureusement, il existe plusieurs décodeurs disponibles gratuitement. Dans cette servlet, nous avons choisi d'utiliser notre propre classe `com.oreilly.servlet.Base64Decoder`, non montrée ici mais disponible à *http://www.servlets.com*, ainsi que `com.oreilly.servlet.Base64Encoder`. Vous pouvez trouver les détails du codage Base64 dans la RFC 1521 à l'adresse *http://www.ietf.org/rfc/rfc1521.txt*.[1]

Exemple 8-8. Sécurité dans une servlet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.Base64Decoder;

public class CustomAuth extends HttpServlet {
```

1. Vous pouvez aussi utiliser la classe `sun.misc.BASE64Decoder` qui accompagne le JDK. Être dans la hiérarchie `sun.*` signifie que la classe n'est pas supportée et qu'elle peut changer, mais cela signifie aussi qu'elle est certainement sur votre système.

Exemple 8-8. Sécurité dans une servlet (suite)

```
Hashtable users = new Hashtable();

public void init(ServletConfig config) throws ServletException {
  super.init(config);

  // Les noms et les mots de passe sont sensibles à la casse !
  users.put("Wallace:cheese",     "allowed");
  users.put("Gromit:sheepnapper", "allowed");
  users.put("Penguin:evil",       "allowed");
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException {
  res.setContentType("text/plain");
  PrintWriter out = res.getWriter();

  // Obtenir l'en-tête Authorization.
  String auth = req.getHeader("Authorization");

  // Acceptons-nous cet utilisateur ?
  if (!allowUser(auth)) {
    // Non autorisé, alors l'indiquer.
    res.setHeader("WWW-Authenticate", "BASIC realm=\"users\"");
    res.sendError(res.SC_UNAUTHORIZED);
    // On pourrait lui proposer de s'ajouter à la liste des utilisateurs
    // autorisés.
  }
  else {
    // Autorisé, on lui montre donc tous nos secrets.
    out.println("Information top secrète");
  }
}

// Cette méthode compare les informations de l'utilisateur envoyées dans
// l'en-tête Authorization à la base de données des utilisateurs maintenue
// dans le Hashtable des utilisateurs.
protected boolean allowUser(String auth) throws IOException {
  if (auth == null) return false;  // Pas d'authentification.

  if (!auth.toUpperCase().startsWith("BASIC "))
    return false;  // Seul BASIC est supporté.

  // Obtenir le mot de passe codé, venant après "BASIC".
  String userpassEncoded = auth.substring(6);

  // Le décoder, en utilisant tout décodeur en base 64 (nous avons choisi
  // com.oreilly.servlet).
  String userpassDecoded = Base64Decoder.decode(userpassEncoded);
```

Exemple 8-8. Sécurité dans une servlet (suite)

```
    // Vérifier dans la liste si cet utilisateur est "autorisé".
    if ("allowed".equals(users.get(userpassDecoded)))
      return true;
    else
      return false;
  }
}
```

Bien que l'on demande au serveur web d'accorder l'accès à cette servlet à tout client, la servlet n'envoie sa sortie top secrète qu'aux utilisateurs qu'elle reconnaît. Avec quelques modifications, elle pourrait autoriser tout utilisateur avec un motif de mot de passe de confiance. Ou, comme le FTP anonyme, elle pourrait autoriser le nom d'utilisateur « anonymous » avec une adresse e-mail pour mot de passe.

L'habilitation personnalisée peut être utilisée pour autre chose que la restriction des accès à une servlet. Si nous ajoutions cette logique à notre servlet `ViewResource`, nous pourrions implémenter une politique d'accès personnalisée à un ensemble entier de fichiers ; on pourrait écrire une règle de correspondance de préfixe d'URL afin que la servlet `ViewResource` sécurisée serve une hiérarchie complète de répertoires contenant des fichiers protégés. Si l'on créait une sous-classe spéciale de `HttpServlet` et y ajoutait cette logique, nous pourrions facilement restreindre l'accès à chaque servlet dérivée de cette sous-classe. Voici notre point de vue : avec l'habilitation personnalisée, les limitations de la politique de sécurité du serveur ne limitent pas les possibilités d'implémentation des politiques de sécurité de ses servlets.

Habilitation personnalisée à base de formulaire

Les servlets ont aussi la possibilité d'effectuer une habilitation personnalisée reposant sur un formulaire. Avec cette technique, une application web peut utiliser une page d'ouverture de session HTML élégante, avec l'avantage que toute politique de sécurité est implémentable, le tout de façon totalement portable. Par exemple, plusieurs applications bancaires ont besoin d'informations en plus du nom d'utilisateur et du mot de passe pour ouvrir une session ; certaines demandent un numéro de compte, un mot de passe et un numéro d'identification personnel (PIN — *personal identification number*). Une telle ouverture de session peut être accomplie à l'aide de `FORM <auth-method>` mais aussi avec une habilitation personnalisée à base de formulaire. Le prix à payer est une complexité supplémentaire car les détails de l'ouverture de session à base de formulaire doivent être pris en charge manuellement.

Les étapes sont assez évidentes. Tout d'abord, nous avons besoin de la page d'ouverture de session. Elle est écrite comme tout autre formulaire HTML. L'*exemple 8-9* donne un fichier *login.html* qui génère le formulaire montré à la *figure 8-3*.

Exemple 8-9. Le fichier login.html

```
<HTML>
<TITLE>Ouverture de session</TITLE>
<BODY>
<FORM ACTION=/servlet/LoginHandler METHOD=POST>
```

Exemple 8-9. Le fichier login.html (suite)

```
<CENTER>
<TABLE BORDER=0>
<TR><TD COLSPAN=2>
<P ALIGN=CENTER>
Bonjour !<br>
Veuillez entrer votre numéro de compte,<br>
votre mot de passe et votre PIN.
</TD></TR>

<TR><TD>
<P ALIGN=RIGHT><B>Compte :</B>
</TD>
<TD>
<P><INPUT TYPE=TEXT NAME="account" VALUE="" SIZE=15>
</TD></TR>

<TR><TD>
<P ALIGN=RIGHT><B>Mot de passe :</B>
</TD>
<TD>
<P><INPUT TYPE=PASSWORD NAME="password" VALUE="" SIZE=15>
</TD></TR>

<TR><TD>
<P ALIGN=RIGHT><B>PIN :</B>
</TD>
<TD>
<P><INPUT TYPE=PASSWORD NAME="pin" VALUE="" SIZE=15>
</TD></TR>

<TR><TD COLSPAN=2>
<CENTER>
<INPUT TYPE=SUBMIT VALUE="  OK   ">
</CENTER>
</TD></TR>
</TABLE>
</FORM>
</BODY></HTML>
```

La *figure 8-3* montre le formulaire.

Ce formulaire demande au client son numéro de compte, son mot de passe et son PIN. Il soumet ensuite cette information à la servlet `LoginHandler` qui valide l'ouverture de session. Nous verrons le code de `LoginHandler` bientôt, mais nous devons d'abord nous demander « Quand ce client va-t-il voir cette page d'ouverture de session ? ». Il est clair qu'il peut aller à cette page directement, peut-être en suivant un lien sur la page d'accueil du site. Mais que se passe-t-il s'il essaie d'accéder directement à une ressource protégée sans passer par la page de connexion ? Dans ce cas, il doit être redirigé vers cette page d'ouverture de session et, après une connexion réussie, être à nouveau dirigé vers

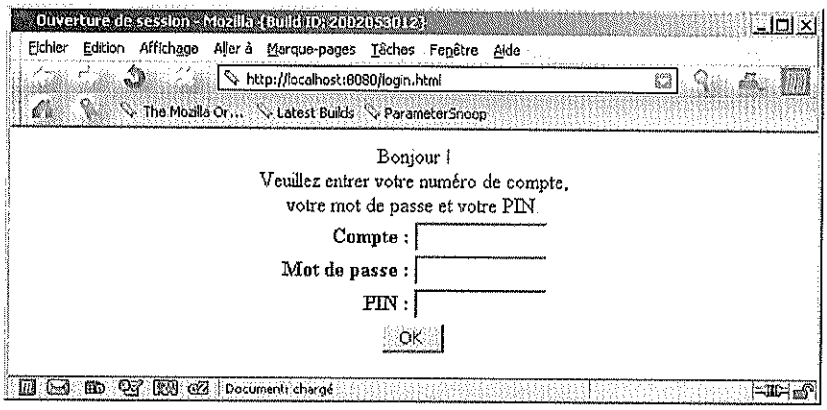


Figure 8-3. Un formulaire d'ouverture de session agréable

la cible de départ. Le processus doit fonctionner aussi simplement que lorsque le navigateur ouvre une fenêtre de dialogue, excepté que dans ce cas le navigateur ouvre une page intermédiaire.

L'*exemple 8-10* montre une servlet qui implémente ce comportement. Elle n'envoie ses données secrètes que si l'objet de session du client indique que sa connexion a déjà été effectuée. Si ce n'est pas le cas, la servlet conserve l'URL demandée dans sa session pour l'utiliser plus tard, et le redirige ensuite vers la page d'ouverture de session

Exemple 8-10. Une ressource protégée

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ProtectedResource extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    // Obtenir la session.
    HttpSession session = req.getSession();

    // La session indique-t-elle que cet utilisateur s'est déjà connecté ?
    Object done = session.getAttribute("logon.isDone");  // Objet marqueur.
    if (done == null) {
      // Pas de logon.isDone signifie qu'il ne s'est pas connecté. Sauver
      // l'URL de la requête comme vraie cible et rediriger vers la page
      // d'ouverture de session.
      session.setAttribute("login.target",
                           HttpUtils.getRequestURL(req).toString());
      res.sendRedirect("/login.html");
```

Exemple 8-10. Une ressource protégée (suite)

```
      return;
    }

    // Si l'on arrive ici c'est que l'utilisateur s'est connecté et qu'il
    // peut voir les choses.
    out.println("Des manuscrits O'Reilly non publiés vous attendent !");
  }
}
```

Cette servlet examine si le client s'est déjà connecté en vérifiant s'il existe dans sa session un objet de nom `logon.isDone`. Si c'est le cas, la servlet sait que le client s'est déjà connecté et lui permet de voir les informations secrètes. Si ce n'est pas le cas, le client n'a pas dû ouvrir sa session, la servlet sauve alors l'URL demandée sous le nom `login.target` et redirige ensuite le client vers la page d'ouverture de session. Avec l'habilitation personnalisée à base de formulaire, toutes les ressources protégées (ou les servlets qui les servent) doivent implémenter ce comportement. L'héritage, ou une classe utilitaire, peut simplifier cette tâche.

Maintenant, passons au gestionnaire d'ouverture de session. Après que le client a entré ses informations dans le formulaire, les données sont envoyées à la servlet `LoginHandler` montrée à l'*exemple 8-11*. Cette servlet contrôle la validité du numéro de compte, du mot de passe et du PIN. Si ce contrôle du client échoue, on lui indique que l'accès est refusé. Sinon, ce fait est enregistré dans son objet de session et il est immédiatement redirigé vers la cible originelle

Exemple 8-11. Gérer une ouverture de session

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginHandler extends HttpServlet {

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir le numéro de compte, le mot de passe et le PIN de l'utilisateur.
    String account = req.getParameter("account");
    String password = req.getParameter("password");
    String pin = req.getParameter("pin");

    // Contrôler leur validité.
    if (!allowUser(account, password, pin)) {
      out.println("<HTML><HEAD><TITLE>Accès refusé</TITLE></HEAD>");
      out.println("<BODY>Vos informations d'ouverture de session " +
                  "ne sont pas valides.<BR>");
      out.println("Voulez-vous <A HREF=\"/login.html\">recommencer</A> ?");
```

Exemple 8-11. Gérer une ouverture de session (suite)

```
      out.println("</BODY></HTML>");
    } else {
      // Connexion valide. Le noter dans l'objet de session.
      HttpSession session = req.getSession();
      session.setAttribute("logon.isDone", account); // Objet marqueur.

      // Essayer de rediriger le client vers sa destination originelle.
      try {
        String target = (String) session.getAttribute("login.target");
        if (target != null) {
          res.sendRedirect(target);
          return;
        }
      }
      catch (Exception ignored) { }

      // Redirection impossible. Rediriger vers la page d'accueil du site.
      res.sendRedirect("/");
    }
  }

  protected boolean allowUser(String account, String password, String pin) {
    return true;  // Faire confiance à tout le monde.
  }
}
```

Le contrôle de validité de cette servlet est plutôt simple : il suppose que tout certificat de l'utilisateur est valide. Les choses sont ainsi plus simples et nous pouvons nous concentrer sur le comportement de la servlet lorsque la connexion réussit. La servlet enregistre le numéro de compte (tout objet bizarre fera l'affaire) de l'utilisateur dans la session du client sous le nom `logon.isDone`, comme marqueur indiquant à toutes les ressources que le client est OK. Elle redirige ensuite le client vers la cible de départ sauvegardée dans `login.target`, l'envoyant là où il voulait d'abord aller. Si cela échoue pour une raison ou pour une autre, la servlet redirige l'utilisateur vers la page d'accueil du site.

Certificats numériques

Les applications réelles demandent un niveau de sécurité plus élevé que celui fourni par l'authentification de base ou à base de formulaire. Elles ont aussi besoin de garantir la confidentialité et l'intégrité, ainsi qu'une authentification plus fiable. La technologie des certificats numériques est là pour cela.

Le concept clé est la cryptographie à clé publique. Dans un système de cryptographie à clé publique, chaque participant possède deux clés utilisées pour crypter et décrypter l'information. L'une est la clé publique, qui est distribuée librement. L'autre est la clé privée, qui est gardée secrète. Les clés sont liées mathématiquement, mais l'une ne peut être dérivée de l'autre. Par exemple, supposons que Jason veuille envoyer un message

secret à Will. Il prend la clé publique de Will et l'utilise pour crypter le message. Lorsque Will reçoit le message, il utilise sa clé privée pour le décrypter. Toute personne interceptant le message en transit est confrontée à un charabia indéchiffrable.

Les schémas de cryptage à clé publique existent depuis plusieurs années et ils sont plutôt bien développés. La plupart sont basés sur l'algorithme breveté RSA développé par Ron Rivest, Adi Shamir et Leonard Adelman.[2] RSA utilise des nombres premiers très grands pour générer une paire de clés asymétriques (chaque clé peut décoder les messages codés avec l'autre clé). Les clés individuelles varient en longueur, généralement exprimée en nombre de bits qui composent la clé. 1 024 ou 2 048 bits sont adaptés aux communications RSA sûres.

Les clés étant longues, il n'est pas facile à un utilisateur d'en saisir une dans son navigateur à chaque requête. À la place, les clés sont stockées sur le disque sous la forme de certificats numériques et les clés privées sont stockées sur le disque protégées par une *expression de passe*. Les certificats numériques peuvent être générés par des logiciels comme le package PGP ou ils peuvent provenir de parties tierces. Les fichiers de certificats eux-mêmes sont chargés par les applications les plus exigeantes en sécurité, comme les serveurs, les navigateurs et les logiciels de courrier électronique.

La cryptographie à clé publique résout le problème de la confidentialité car la communication est cryptée. Elle résout aussi le problème d'intégrité : Will sait que le message qu'il reçoit n'a pas été altéré car il est correctement décodé. Et quid de l'authentification ? Will ne sait pas si c'est bien Jason qui a envoyé le message. C'est là qu'entrent en jeu les signatures numériques. Les clés publiques et privées étant asymétriques, Jason peut d'abord utiliser sa clé privée pour coder un message et ensuite utiliser la clé publique de Will pour le coder une fois encore. Lorsque Will reçoit le message, il le décode d'abord avec sa clé privée et ensuite avec la clé publique de Jason. Jason ne pouvant coder ses messages qu'avec sa clé privée, messages qui ne peuvent décodés qu'avec sa clé publique, Will sait que le message a vraiment été envoyé par Jason.

C'est différent des systèmes à clés symétriques plus simple, où une seule clé est utilisée le codage et pour le décodage. Bien que les clés asymétriques aient l'avantage significatif d'autoriser des communication sûres sans imposer un canal sûr, elles ont cependant l'inconvénient de demander beaucoup plus de puissance de calcul. Le compromis fait par beaucoup de systèmes de chiffrement est d'utiliser des clés publique et privée asymétriques pour s'identifier et ensuite d'échanger confidentiellement une autre clé symétrique pour chiffrer les échanges. Ces clés symétriques sont généralement basées sur DES (*Data Encryption Standard*).

Les restrictions d'exportation établies par le gouvernement des États-Unis limitent actuellement la taille d'une clé à 56 bits (autour de 72 quadrillions de clés possibles). Les messages chiffrés avec une clé sur 56 bits sont difficiles à décoder, mais ce n'est pas impossible ; des machines spécialisées ont été utilisés pour décoder de tels messages en quelques heures. Cependant, au sein des États-Unis, beaucoup de systèmes utilisent des clés DES sur 128 bits (autour de $3{,}40282 \times 10^{38}$ clés possibles). Puisqu'il n'existe pas de façon connue de décoder un message crypté par DES autre que par essai/erreur, les messages envoyés en utilisant des clés longues sont très, très sûrs.

2. United States Patent No. 4,405,829. Il a expiré le 20 septembre 2000.

Il reste un dernier problème : comment un utilisateur sait-il qu'un autre utilisateur est celui qu'il prétend être ? Jason et Will se connaissent, Will fait donc confiance à la clé publique que Jason lui donne en personne[3]. D'un autre côté, si Lisa veut donner à Jason sa clé publique, mais que Jason et Lisa ne se sont jamais rencontrés, il n'y a aucune raison pour que Jason croit que Lisa n'est pas en fait Mark. Mais, si nous supposons que Will connaît Lisa, Will peut utiliser sa clé privée pour signer la clé publique de Lisa. Ensuite, lorsque Jason obtient la clé, il détecte que Will, à qui il fait confiance, souhaite confirmer l'identité de Lisa. Ces présentations sont parfois appelées *réseau de confiance*.

Dans le monde réel, cette confirmation à trois est généralement gérée par une autorité certifiée spécialement créée, comme VeriSign Corporation. VeriSign étant une organisation bien connue avec une clé publique bien connue, les clés vérifiées et signées par VeriSign peuvent être considérées comme fiables, du moins tant que VeriSign a bien la preuve de l'identité du receveur. VeriSign propose différentes classes d'identifiants numériques, chacune avec un niveau de confiance et un prix croissant. Vous pouvez obtenir un identifiant de classe 1 en remplissant simplement un formulaire sur le site web de VeriSign et en recevant un e-mail. Les classes de plus haut niveau sont individuellement vérifiées par les employés de VeriSign, par des contrôles et des services d'investigation.

Lors du choix d'une autorité de certification, il est important de choisir une société ayant une forte présence sur le marché. Les certificats de VeriSign, par exemple, sont inclus dans Netscape Navigator et Microsoft Internet Explorer, chaque utilisateur de l'Internet peut donc leur faire confiance et les accepter. Les sociétés suivantes fournissent des services de certification :

- VeriSign (*http://www.verisign.com*) ;
- Thawte Consulting (*http://www.thawte.com*) ;
- Entrust Technologies (*http://www.entrust.com*).

Pour des informations plus abstraites sur les certificats numériques, nous vous recommandons le livre *Understanding Digital Signatures* de Gail L. Grant (Mc-Graw Hill), qui fournit une excellente introduction sur ce sujet adaptée aux programmeurs et non-programmeurs. Sur la cryptographie et son lien avec Java, nous vous recommandons de lire *Java Cryptography* de Jonathan Knudsen (O'Reilly & Associates).

Secure Sockets Layer (SSL)

Le protocole SSL (*Secure Sockets Layer*) se place entre le protocole applicatif (ici HTTP) et le protocole de transport (pour l'Internet il s'agit presque exclusivement de TCP/IP). Il s'occupe des détails de la gestion de la sécurité en utilisant la cryptographie à clé publique pour échanger des clés symétriques qui chiffrent toutes les communications entre un client et un serveur. SSL a été introduit par Netscape dans Netscape Navigator 1. Il est devenu depuis le standard *de facto* pour les communications en ligne sûres et il forme les bases du protocole TLS (*Transport Layer Security*) actuellement en cours de déve-

3. Pour être franc, les gens ne se rencontrent presque jamais dans des ruelles sombres pour échanger leur clé publique. Ils les échangent numériquement (peut-être par e-mail) et comparent simplement en personne une petite empreinte de la clé.

loppement par l'IETF (*Internet Engineering Task Force*). Pour plus d'informations sur TLS, allez voir à *http://www.ietf.org/rfc/rfc2246.txt*.

SSL Version 2.0, la première version à avoir été largement acceptée, inclut uniquement le support des certificats sur le serveur. Elle offre l'authentification du serveur, la confidentialité et l'intégrité. Voici comment elle fonctionne :

1. Un utilisateur se connecte à un site sûr en utilisant le protocole HTTPS (HTTP sur SSL). (Vous pouvez détecter les sites utilisant le protocole HTTPS car leurs URL commencent par *https:* au lieu de *http:*.)
2. Le serveur signe sa clé publique avec sa clé privée et l'envoie au navigateur.
3. Le navigateur utilise la clé publique du serveur pour vérifier que personne qui l'a signée est bien celle à qui elle appartient.
4. Le navigateur effectue un contrôle pour savoir si une autorité de certification de confiance a signé la clé. Si ce n'est pas le cas, le navigateur demande à l'utilisateur s'il peut faire confiance à la clé et continue en fonction de la réponse.
5. Le client génère une clé symétrique (DES) pour la session, qui est chiffrée avec la clé publique du serveur et lui est retournée. Cette nouvelle clé est utilisée pour chiffrer toutes les transactions à venir. La clé symétrique est utilisée à cause du coût en calcul élevé des systèmes de cryptographie à clés publiques.

Tout cela est complètement transparent pour les servlets et les développeurs. Vous devez juste obtenir le bon certificat du serveur, l'installer et configurer correctement votre serveur. Les informations transférées entres les servlets et les clients sont maintenant chiffrées. Et voilà, sécurité !

Authentification SSL du client

Notre boîte à outils pour la sécurité inclut maintenant une cryptographie et une authentification du serveur solides, mais une authentification du client faible. Bien sûr, utiliser SSL 2.0 nous place dans de meilleures conditions car les serveurs équipés de SSL peuvent utiliser les méthodes d'authentification de base présentées au début de ce chapitre sans s'occuper des problèmes d'écoute. Cependant, nous n'avons toujours pas la preuve de l'identité du client. Après tout, n'importe qui peut avoir deviné ou obtenu le nom d'utilisateur et le mot de passe du client.

SSL 3.0 pallie ce problème en fournissant le support des certificats clients. Ils sont du même type que ceux utilisés par les serveurs, mais ils sont réservés aux clients. SSL 3.0 avec l'authentification du client fonctionne de la même façon que SSL 2.0, sauf qu'après l'authentification du serveur par le client, le serveur demande le certificat du client. Celui-ci envoie son certificat signé et le serveur effectue le même processus d'authentification que le client, comparant le certificat client à une bibliothèque de certificats existants (ou simplement en stockant le certificat pour identifier le client lors d'une prochaine visite). Pour des raisons de sécurité, beaucoup de navigateurs demandent à l'utilisateur d'entrer son mot de passe avant d'envoyer le certificat.

Une fois le client authentifié, le serveur peut autoriser les accès aux ressources protégées telles que les servlets ou les fichiers, comme dans le cas de l'authentification HTTP. Le processus est transparent, sans que l'utilisateur soit importuné. Il fournit aussi un ni-

veau additionnel d'authentification car le serveur sait que le client possédant le certificat John Smith est vraiment John Smith (et il peut savoir quel John Smith en lisant son certificat unique). Les inconvénients des certificats clients sont que les utilisateurs doivent obtenir et installer des certificats signés, les serveurs doivent maintenir une base de données de toutes les clés publiques acceptées et ils doivent supporter SSL 3.0. Au moment de l'écriture de ce livre, c'est le cas de la plupart.

Configurer la sécurité SSL

Une application web nécessitant une sécurité SSL (HTTPS) peut l'indiquer au serveur à l'aide de son descripteur de déploiement. Quand SSL est nécessaire, le fichier *web.xml* doit être modifié de sorte que la balise `<security-constraint>` contienne une balise `<user-data-constraint>` indiquant les contraintes de sécurité. L'*exemple 8-12* montre cela.

Exemple 8-12. Cette collection demande une connexion sûre

```
<!-- ...etc... -->

<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            SecretProtection
        </web-resource-name>
        <url-pattern>
            /servlet/SalaryServer
        </url-pattern>
        <url-pattern>
            /servlet/secret
        </url-pattern>
        <http-method>
            GET
        </http-method>
        <http-method>
            POST
        </http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>
            directeur
        </role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>
            CONFIDENTIAL        <!-- INTEGRAL ou CONFIDENTIAL -->
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>

<!-- ...etc... -->
```

L'ajout de <user-data-constraint> indique non seulement que cet ensemble de ressources impose à l'utilisateur de faire partie du rôle directeur, mais aussi que les utilisateurs doivent se connecter à l'aide d'une connexion CONFIDENTIAL.

La balise <transport-guarantee> peut prendre deux valeurs : INTEGRAL ou CONFIDENTIAL. INTEGRAL impose que les données doivent être garanties contre toute modification pendant le transport. CONFIDENTIAL impose que les données ne puissent être lues par une partie tierce pendant le transport. La garantie CONFIDENTIAL implique la garantie INTEGRAL. C'est au serveur de décider des algorithmes de chiffrement adaptés à INTEGRAL et à CONFIDENTIAL. La plupart des algorithmes SSL sont adaptés aux deux, bien que le serveur puisse considérer que le chiffrement DES sur 56 bits soit suffisant pour INTEGRAL (car pour décoder le message, il ne doit pas avoir été modifié) mais insuffisant pour CONFIDENTIAL (car il peut être facilement craqué).[4] En pratique, il y a aujourd'hui sur le Web très peu de différence entre les deux, et CONFIDENTIAL est la garantie standard.

Les balises <auth-constraint> et <user-data-constraint> peuvent exister de façon indépendante ou non. Par exemple, une page de traitement d'une carte de crédit demandera des communications CONFIDENTIAL mais elle peut rester visible à tous les utilisateurs.

Configurer l'authentification SSL

Au fur et à mesure que SSL 3.0 gagne en popularité, de plus en plus de sites veulent authentifier les utilisateurs avec leur certificat client. Cela fournit une méthode automatique et sûre pour déterminer l'identité du client. En fonction du niveau de confiance associé au certificat du client, ce mécanisme d'authentification peut être considéré comme suffisamment sûr pour supporter des contrats légaux et même le vote en ligne.

Ceux qui ont lu les commentaires dans le fichier *web.xml* précédents doivent être capables de dire que l'authentification basée sur les certificats client peut être spécifiée par une modification de la balise <login-config> :

```
<login-config>
    <auth-method>
        CLIENT-CERT <!-- Le client doit être reconnu avec un -->
                    <!-- certificat X.509 -->
    </auth-method>
</login-config>
```

Cela indique au serveur que toutes les authentifications de cette application web se font avec des certificats client au lieu des autres solutions de base ou par formulaire. le client ne verra jamais de page d'ouverture de session, même si le navigateur lui demande un mot de passe pour déverrouiller son certificat avant de l'envoyer au serveur. Si le navigateur n'a pas de certificat client, l'accès est refusé.

4. Une connexion client/serveur au-dessus d'un réseau privé virtuel VPN (*Virtual Private Network*) sûr peut satisfaire la contrainte CONFIDENTIAL, même si SSL n'est pas utilisé.

Retrouver l'information d'authentification SSL

Comme avec les authentifications de base et digest, tous les détails de SSL sont gérés par le serveur et sont donc transparents pour les servlets. Autrement dit, une servlet n'a rien à faire pour fonctionner dans un serveur sûr ! Cependant, il est quelquefois possible qu'une servlet retrouve l'information d'authentification SSL pertinente. Par exemple, la méthode `isSecure()` permet à une servlet de savoir si la connexion au client est sûre :

```
public boolean ServletRequest.isSecure()
```

Cette méthode retourne `true` si le serveur juge la connexion sûre. Les implications de « sûre » sur le chiffrement dépend de l'implémentation du serveur ; consultez la documentation de votre serveur pour les détails. Malheureusement, une servlet ne dispose d'aucun moyen standard pour demander l'algorithme de chiffrement utilisé pour la connexion, ni même la taille (40, 56, 128 bits) de la clé symétrique utilisée pour le chiffrement. Un serveur a le choix de rendre cette information disponible dans un attribut de requête, mais il n'y a pas encore de nom d'attribut normalisé. Cette fonctionnalité est attendue dans l'API Servlet 2.3 avec les noms d'attributs `javax.servlet.request.cipher_suite` et `javax.servlet.request.key_size`.

Quand le client a été authentifié à l'aide de `CLIENT-CERT`, son nom de principal est disponible *via* la méthode `getUserPrincipal()` présentée précédemment. Le nom de principal vient du champ Distinguished Name du certificat.

Enfin, chaque fois qu'un certificat client est envoyé au serveur (ce qui peut se produire pendant le processus de présentation normal de SSL 3.0, même lorsque `CLIENT-CERT` n'est pas spécifié), une servlet peut retrouver le certificat du client dans un attribut de requête :

```
java.security.cert.X509Certificate cert =
  (java.security.cert.X509Certificate)
  req.getAttribute("javax.servlet.request.X509Certificate");
```

Pour tout serveur s'exécutant sur J2SE 1.2 (JDK 1.2) ou supportant J2EE 1.2, l'attribut de requête `javax.servlet.request.X509Certificate` retourne un objet `java.security.cert.X509Certificate` représentant un certificat X.509v3 (voir la documentation de votre serveur pour les instructions de configuration). Les serveurs utilisant le JDK 1.1 et qui ne veulent pas être compatibles avec J2EE 1.2 ne sont pas obligés de supporter cet attribut (car le JDK 1.1 n'inclut pas le paquetage `java.security.cert`). La servlet de *exemple 8-13* affiche le chaîne de certificat du client, si elle est disponible.

Exemple 8-13. Examiner les certificats du client

```
import java.io.*;
import java.util.*;
import java.security.cert.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class X509Snoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
```

Exemple 8-13. Examiner les certificats du client (suite)

```
    PrintWriter out = res.getWriter();

    X509Certificate[] certs = (X509Certificate[])
      req.getAttribute("javax.servlet.request.X509Certificate");
    if (certs != null) {
      for (int i = 0; i < certs.length; i++) {
        out.println("Certificat client [" + i + "] = "
                    + certs[i].toString());
      }
    }
    else {
      if ("https".equals(req.getScheme())) {
        out.println("Requête HTTPS, mais aucun " +
                    "certificat client n'est disponible");
      }
      else {
        out.println("Requête non-HTTPS, aucun " +
                    "certificat client n'est donc disponible");
      }
    }
  }
}
```

Le X509Certificate retourné peut être pris à part et utilisé pour contrôler la validité, l'émetteur, le numéro de série, la signature, etc. L'affichage de chaque certificat sous forme d'un String donne la sortie montrée à l'*exemple 8-14*. Le premier certificat est la clé publique de l'utilisateur. Le second est la signature de VeriSign qui confirme l'authenticité de la première signature.

Exemple 8-14. La chaîne de certificats X.509 de Ramesh Mandava

```
Certificat client [0] = [
[
  Version: V3
  Subject: EmailAddress=rmandava@talentportal.com, CN=Ramesh Babu Mandava,
OU=Digital ID Class 1 -
Netscape, OU=Persona Not Validated, OU="www.verisign.com/repository/RPA Incorp.
by Ref.,LIAB.LTD(c)98",
OU=VeriSign Trust Network, O="VeriSign, Inc."
  Signature Algorithm: MD5withRSA, OID = 1.2.840.113549.1.1.4

  Key:  com.sun.net.ssl.internal.ssl.JSA_RSAPublicKey@5b5870e3
  Validity: [From: Tue Oct 10 17:00:00 PDT 2000,
              To: Sun Dec 10 15:59:59 PST 2000]
  Issuer: CN=VeriSign Class 1 CA Individual Subscriber-Persona Not Validated,
OU="www.verisign.com/repository/RPA Incorp. By Ref.,LIAB.LTD(c)98", OU=VeriSign
Trust Network,
O="VeriSign, Inc."
  SerialNumber: [    1ef11638 5ab8aaa1 bfa2b1b3 c0fb9cd9 ]
```

Exemple 8-14. La chaîne de certificats X.509 de Ramesh Mandava (suite)

```
Certificate Extensions: 4
[1]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false
NetscapeCertType [
   SSL client
]

[2]: ObjectId: 2.5.29.32 Criticality=false
Extension unknown: DER encoded OCTET string =
0000: 04 3D 30 3B 30 39 06 0B   60 86 48 01 86 F8 45 01  .=0;09..`.H...E.
0010: 07 01 08 30 2A 30 28 06   08 2B 06 01 05 05 07 02  ...0*0(..+......
0020: 01 16 1C 68 74 74 70 73   3A 2F 2F 77 77 77 2E 76  ...https://www.v
0030: 65 72 69 73 69 67 6E 2E   63 6F 6D 2F 72 70 61     erisign.com/rpa


[3]: ObjectId: 2.5.29.31 Criticality=false
Extension unknown: DER encoded OCTET string =
0000: 04 2C 30 2A 30 28 A0 26   A0 24 86 22 68 74 74 70  .,0*0(.&.$."http
0010: 3A 2F 2F 63 72 6C 2E 76   65 72 69 73 69 67 6E 2E  ://crl.verisign.
0020: 63 6F 6D 2F 63 6C 61 73   73 31 2E 63 72 6C        com/class1.crl


[4]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
CA:false
PathLen: undefined
]

]
  Algorithm: [MD5withRSA]
  Signature:
0000: 5E EC 5C F9 96 D5 3F F6   19 8B 66 0A 46 DE 02 FC  ^.\...?...f.F...
0010: 52 4E 32 70 5E DA 8B 92   43 F4 19 51 C3 A3 36 7D  RN2p^...C..Q..6.
0020: 02 4A 5B 35 B6 76 05 F8   FE C0 4F D7 9C B1 5B BA  .J[5.v....O...[.
0030: EE 38 A7 98 C5 57 A7 6B   86 B9 B2 A1 4F 25 5F FF  .8...W.k....O%_.
0040: 0B 19 54 86 D7 14 7A F7   97 A1 E8 E7 D3 89 75 B0  ..T...z.......u.
0050: 72 4F 4B 77 E4 56 5D B2   40 D2 7E 69 26 77 DD F1  rOKw.V].@..i&w..
0060: E6 31 3D F2 EF 5A 11 22   78 23 47 C2 D6 ED DD 14  .1=..Z."x#G.....
0070: 2F E9 2E 46 73 D9 20 72   BF 9B 6C 04 12 0D 68 C7  /..Fs. r..l...h.

]
Certificate client [1] = [
[
  Version: V3
  Subject: CN=VeriSign Class 1 CA Individual Subscriber-Persona Not Validated,
OU="www.verisign.com/repository/RPA Incorp. By Ref.,LIAB.LTD(c)98", OU=VeriSign
Trust Network,
O="VeriSign, Inc."
  Signature Algorithm: MD2withRSA, OID = 1.2.840.113549.1.1.2
```

Exemple 8-14. La chaîne de certificats X.509 de Ramesh Mandava (suite)

```
  Key:  com.sun.net.ssl.internal.ssl.JSA_RSAPublicKey@9ae870e3
  Validity: [From: Mon May 11 17:00:00 PDT 1998,
              To: Mon May 12 16:59:59 PDT 2008]
  Issuer: OU=Class 1 Public Primary Certification Authority, O="VeriSign,
Inc.", C=US
  SerialNumber: [    d2762e8d 140c3d7d b2a8255d afee0d75 ]

Certificate Extensions: 4
[1]: ObjectId: 2.16.840.1.113730.1.1 Criticality=false
NetscapeCertType [
   SSL CA
   S/MIME CA
]

[2]: ObjectId: 2.5.29.32 Criticality=false
Extension unknown: DER encoded OCTET string =
0000: 04 40 30 3E 30 3C 06 0B   60 86 48 01 86 F8 45 01  .@0>0<..`.H...E.
0010: 07 01 01 30 2D 30 2B 06   08 2B 06 01 05 05 07 02  ...0-0+..+......
0020: 01 16 1F 77 77 77 2E 76   65 72 69 73 69 67 6E 2E  ...www.verisign.
0030: 63 6F 6D 2F 72 65 70 6F   73 69 74 6F 72 79 2F 52  com/repository/R
0040: 50 41                                              PA


[3]: ObjectId: 2.5.29.15 Criticality=false
KeyUsage [
  Key_CertSign
  Crl_Sign
]

[4]: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
CA:true
PathLen:0
]

]
  Algorithm: [MD2withRSA]
  Signature:
0000: 88 B8 37 3B DD DA 94 37   00 AD AA 9F E1 81 01 71  ..7;...7.......q
0010: 1E 92 6A 6D 2F F6 F1 9D   D3 CA 64 38 DC 1B 98 0C  ..jm/.....d8....
0020: 07 86 5B 85 15 6A 0F B9   49 85 A4 95 F1 17 7D 67  ..[..j..I......g
0030: B4 7F 2D 2C DD 9A 42 9E   C3 3E B4 8E AA E5 0B 06  ..-,..B..>......
0040: DE F2 56 2A FA 33 C7 BE   19 D7 53 4C C3 BD C8 E3  ..V*.3....SL....
0050: 17 B5 A4 49 42 63 EC C2   A6 17 0F 5D 58 1A 49 3C  ...IBc.....]X.I<
0060: 90 5C 55 A3 65 20 00 FD   18 20 E5 5F 82 A6 B1 A8  .\U.e ... ._....
0070: 92 C5 58 6A C1 8D 03 3C   EB C3 CD 05 A2 90 AE 6E  ..Xj...<.......n

]
```

9

Se connecter aux bases de données

Il est difficile aujourd'hui de trouver un site web professionnel qui n'ait pas une forme de connexion à une base de données. Les webmestres ont connecté des interfaces en ligne à toutes sortes d'anciens systèmes, y compris des bases de données de suivi des colis et des annuaires, ainsi que plusieurs systèmes plus récents comme les messageries en ligne, l'accès à des magasins et les moteurs de recherche. Mais l'interaction web/base de données a un prix : ces types de sites web sont complexes à développer et peuvent souvent montrer de gros problèmes de performance. Pourtant, pour beaucoup de sites web, spécialement les applications intranet, la connexion à une base de données est trop utile pour s'en passer. De plus en plus, les bases de données s'imposent sur le Web.

Ce chapitre présente les bases de données relationnelles, le langage SQL (*Structured Query Language*) utilisé pour les manipuler et l'API JDBC (*Java Database Connectivity*) elle-même. Les servlets, avec leur cycle de vie à longue échéance, et JDBC, une API bien définie de connexion à une base de données quelconque, forment une solution élégante et efficace dont peuvent profiter les webmestres. En fait, les deux auteurs ont commencé à travailler avec les servlets à cause de cette efficacité et cette élégance. Même si ailleurs dans ce livre nous avons supposé que vous étiez familier de Java, ce chapitre commence par une formation rapide à JDBC.

Le plus gros avantage des servlets concernant la connexion aux bases de données est que leur cycle de vie (expliqué au chapitre 3, *Cycle de vie d'une servlet*) les autorise à maintenir des réserves de connexions aux bases de données ouvertes. Une connexion existante peut gagner plusieurs secondes par rapport à un script CGI qui doit établir sa connexion à chaque invocation. La façon exacte de maintenir la connexion dépend de la tâche à exécuter, et ce chapitre montre plusieurs techniques appropriées à différentes tâches.

Un autre avantage des servlets sur les programmes CGI et beaucoup d'autres technologies est que JDBC est indépendant de la base de données. Une servlet écrite pour accéder à une base de données Sybase peut, avec une modification de deux lignes ou un changement dans un fichier de propriétés, accéder à une base de données Oracle (si l'on suppose qu'aucun des appels qu'elle effectue n'est particulier à un fournisseur). En fait, vous pourrez noter que les exemples de ce chapitre sont écrits pour accéder à différentes bases de données, y compris les sources de données ODBC, comme Microsoft Access, Oracle et Sybase.

Servlets dans le tiers intermédiaire

Les servlets, spécialement celles accédant à une base de données, sont souvent placées dans ce que l'on appelle le *tiers intermédiaire*. Un logiciel dans le tiers intermédiaire aide à connecter une extrémité à une autre (une applet à une base de données) et ajoute quelque chose en cours de route.

Une très bonne raison d'ajouter un logiciel intermédiaire entre un client et notre source de données est que le logiciel dans le tiers intermédiaire (ou *middleware*) peut inclure une logique métier. La logique métier transforme les tâches complexes de bas niveau (comme la mise à jour des tables de la base) en des tâches de haut niveau (effectuer une commande), rendant l'opération globale plus simple et plus sûre.

Imaginez une application client qui effectue une commande. Sans le middleware, elle devrait se connecter directement au serveur de base de données qui stocke les commandes pour ensuite modifier les champs de la base pour effectuer la commande. Si le serveur change (en changeant de machine, en modifiant la structure interne des tables, ou en changeant de fournisseur de bases de données), le client peut ne plus fonctionner. Pire encore, si quelqu'un effectue une modification mineure sur le client (volontairement ou non), il est possible que la base de données enregistre des commandes sans d'abord recevoir le paiement ou rejette des entrées parfaitement valides.

Le middleware utilise la logique métier pour rendre le processus de commande abstrait. Il accepte des informations sur la commande (par exemple le nom, l'adresse, l'article, la quantité, le numéro de carte de crédit), contrôle ces informations, vérifie que la carte de crédit est valide et entre les informations dans la base. Si la base de données devait changer, le middleware peut être mis à jour sans changer le client. Même si la base de données des commandes est temporairement remplacée par un journal de commandes utilisant un fichier, le middleware peut afficher la même représentation au client.

Le middleware peut améliorer l'efficacité en répartissant la charge de traitement sur plusieurs serveurs (serveurs de CPU, de bases de données, de fichiers, d'annuaires, etc.). Il peut aussi utiliser plus efficacement la bande passante : au lieu que le client effectue des communications aller-retour incessantes avec le serveur sur une connexion réseau peut-être lente, le client peut indiquer au middleware ce qu'il veut et ce dernier peut effectuer le travail en utilisant une connexion réseau rapide et probablement mettre les connexions dans une réserve.

Sur le Web, le tiers intermédiaire est souvent implémenté par des servlets. Elles offrent une façon pratique de connecter des clients construits avec des formulaires HTML ou des applets à des serveurs. Un client communique ses demandes à la servlet *via* HTTP, et la logique métier de la servlet traite la requête en se connectant au serveur. (Vous aurez plus d'informations sur la communication applet-servlet au chapitre 10, *Communication applet-servlet*.)

— *Suite...* —

Les servlets utilisent parfois un autre tiers intermédiaire derrière le serveur web pour se connecter à une base de données, comme les EJB (*Enterprise JavaBeans*). Si un serveur web envoie un formulaire HTML avec une information de commande à une servlet, celle-ci analyse l'information et effectue un appel RMI à un EJB sur une autre machine responsable du traitement des commandes, par des servlets ou des programmes autonomes. Dans ce cas, ce qui était alors trois tiers est devenu quatre tiers.

Bases de données relationnelles

Dans certains des exemples précédents, nous avons vu des servlets qui utilisent les fichiers du disque local pour stocker leurs données persistantes. L'utilisation d'un fichier plat est parfait pour de petites quantités de données, mais il peut vite devenir ingérable. La simple recherche des données peut devenir un vrai défi : imaginez que vous stockiez le nom, la ville et l'adresse e-mail de chacun de vos clients dans un fichier texte. Cela fonctionne bien pour une société qui démarre, mais que se passe-t-il lorsque vous avez des centaines de milliers de clients et que vous voulez afficher une liste de ceux qui habitent Boston et dont l'adresse e-mail se termine par *aol.com* ?

Une des meilleures solutions à ce problème est un SGBDR (*système de gestion de bases de données relationnelles*). Au plus bas niveau, un SGBDR organise les données dans des tables. Ces tables sont organisées en lignes et colonnes, un peu comme les tableurs. Des lignes et des colonnes particulières dans une table peuvent être mises en relation (d'où le terme *relationnelle*) avec des lignes et des colonnes d'une autre table.

Une table dans une base de données relationnelles peut contenir des informations sur les clients, une autre peut contenir des commandes et une troisième peut contenir des informations sur les articles d'une commande. En incluant des identifiants uniques (disons des numéros de client et des numéros de commande), les commandes de la table des commandes peuvent être liées aux enregistrements des clients et aux composants individuels d'une commande. La *figure 9-1* montre comment cela pourrait se représenter sur un papier.

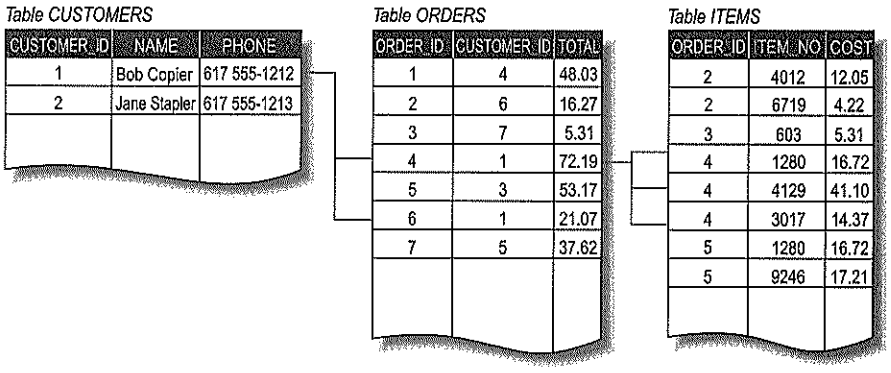


Figure 9-1. Relations entre des tables

Les données des tables peuvent être lues, mises à jour, ajoutées et supprimées en utilisant SQL (*Structured Query Language*), quelquefois aussi appelé le *Standard Query Language*. L'API JDBC de Java introduite dans le JDK 1.1 utilise un sous-ensemble spécifique de SQL connu sous le nom ANSI SQL-2 Entry Level. Contrairement à la plupart des langages de programmation, SQL est déclaratif : vous dites ce que vous voulez et l'interpréteur SQL vous le donne. D'autres langages comme C, C++ et Java sont essentiellement procéduraux, vous devez indiquer les étapes permettant d'effectuer une certaine tâche. SQL, sans être extrêmement complexe, est aussi un sujet trop vaste pour qu'il puisse être étudié en détail ici. Afin que les exemples de ce chapitre soient compréhensibles, voici cependant une petite formation. Pour une initiation plus complète aux bases de données relationnelles et à SQL, nous vous recommandons *SQL pour les nuls* d'Allen Taylor (IDG Books Worldwide), comme livre d'introduction, et *SQL in a Nutshell* de Kevin Kline et Daniel Kline, Ph.D. (O'Reilly & Associates), comme livre de référence.

L'expression SQL la plus simple et la plus commune est l'instruction SELECT, qui interroge la base de données et retourne un ensemble de lignes correspondant à un ensemble de critères de recherche. Par exemple, l'instruction SELECT suivante sélectionne tous les clients de la table CUSTOMERS :

```
SELECT * FROM CUSTOMERS
```

Les mots-clés SQL comme SELECT et FROM et les objets comme CUSTOMERS ne sont pas sensibles à la casse mais ils sont souvent écrits en majuscules. Lorsqu'elle est passée à l'interpréteur SQL*PLUS d'Oracle, cette requête pourrait produire la sortie suivante :

```
CUSTOMER_ID   NAME                          PHONE
------------- ----------------------------- ---------------
1             Bob Copier                    617 555-1212
2             Janet Stapler                 617 555-1213
3             Joel Laptop                   508 555-7171
4             Larry Coffee                  212 555-6525
```

Des instructions plus avancées pourraient restreindre l'interrogation à des colonnes particulières ou inclure des critères limitatifs :

```
SELECT ORDER_ID, CUSTOMER_ID, TOTAL FROM ORDERS
WHERE ORDER_ID = 4
```

Cette instruction sélectionne les colonnes ORDER_ID, CUSTOMER_ID et TOTAL de tous les enregistrements où le champ ORDER_ID est égal à 4. Voici un résultat possible :

```
 ORDER_ID CUSTOMER_ID     TOTAL
--------- ----------- ---------
        4           1     72.19
```

Une instruction SELECT peut aussi relier deux tables ou plus en fonction de la valeur de champs particuliers. Ce peut être une relation un-à-un ou, le plus souvent, une relation un-à-plusieurs, comme un client à plusieurs commandes :

```
SELECT CUSTOMERS.NAME, ORDERS.TOTAL FROM CUSTOMERS, ORDERS
WHERE ORDERS.CUSTOMER_ID = CUSTOMERS.CUSTOMER_ID AND ORDERS.ORDER_ID = 4
```

Cette instruction connecte (ou, dans le jargon base de données, joint) la table CUSTOMERS avec la table ORDERS *via* le champ CUSTOMER_ID. Notez que les deux tables ont ce champ. La requête retourne les informations des deux tables : le nom du client qui a placé la commande 4 et son coût total. Voici un résultat possible :

```
NAME                           TOTAL
------------------------------ ---------
Bob Copier                     72.19
```

SQL est aussi utilisé pour mettre à jour la base de données. Par exemple :

```
INSERT INTO CUSTOMERS (CUSTOMER_ID, NAME, PHONE)
  VALUES (5, "Bob Smith", "555 123-3456")

UPDATE CUSTOMERS SET NAME = "Robert Copier" WHERE CUSTOMER_ID = 1

DELETE FROM CUSTOMERS WHERE CUSTOMER_ID = 2
```

La première instruction crée un nouvel enregistrement dans la table CUSTOMERS, remplissant les champs CUSTOMER_ID, NAME et PHONE avec certaines valeurs. La seconde met à jour un enregistrement existant, en changeant la valeur du champ NAME d'un client spécifique. La dernière supprime tout enregistrement dont CUSTOMER_ID vaut 2. Soyez très prudent avec toutes ces instructions, spécialement avec DELETE. Une instruction DELETE sans clause WHERE supprimera tous les enregistrements de la table !

L'API JDBC

Précédemment, nous avons supposé que vous aviez une connaissance générale des diverses API de Java. Mais comme un programmeur Java même expérimenté peut avoir relativement peu d'expérience avec les base de données, cette section donne une introduction générale à JDBC. Ce chapitre présente principalement la version 1.2 de JDBC, celle qui est la plus largement supportée. À la fin de ce chapitre, nous présentons rapidement JDBC Version 2.0.

Si c'est votre premier pas dans le monde des bases de données, nous vous recommandons fortement de souffler un peu et de prendre un livre sur les bases de données en général et sur les concepts de JDBC. Vous pouvez choisir de lire *JDBC et Java, Guide du programmeur,* de George Reese (Éditions O'Reilly), ou *JDBC Database Access with Java* de Graham Hamilton, Rick Cattell et Maydene Fisher (Addison-Wesley). Une présentation rapide est aussi donnée dans *Java Enterprise in a Nutshell* de David Flanagan et *al.* (Éditions O'Reilly). La spécification officielle de JDBC est disponible en ligne à *http://java.sun.com/products/jdbc.*

JDBC est une API du niveau SQL ; elle vous permet d'exécuter des instructions SQL et de retrouver les résultats, s'il y en a. L'API elle-même est un ensemble d'interfaces et de classes conçues pour effectuer des actions sur toute base de données. La *figure 9-2* montre comment les programmes JDBC interagissent avec les bases de données.

Pilotes JDBC

L'API JDBC, qui se trouve dans le paquetage `java.sql`, ne contient que quelques classes concrètes. La majeure partie de l'API est répartie dans des classes d'interface indépendantes des bases de données qui spécifient des comportements sans fournir de mise en œuvre. Les implémentations sont apportées par des fournisseurs tiers.

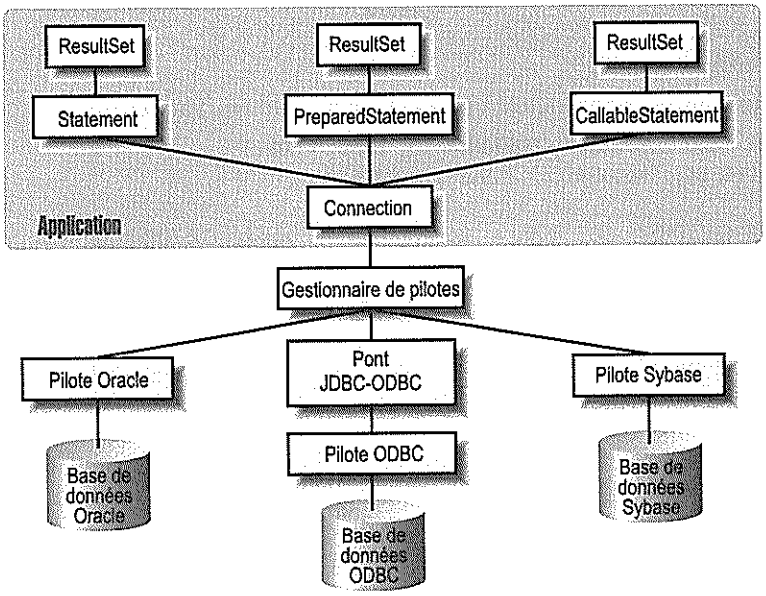


Figure 9-2. Java et les bases de données

On accède à un système de base de données individuel *via* un pilote JDBC spécifique qui implémente l'interface `java.sql.Driver`. Des pilotes existent pour presque tous les SGBDR, mais peu d'entre eux sont gratuits. Sun fournit gratuitement avec le JDK un pilote qui sert de pont entre JDBC et ODBC afin de pouvoir accéder aux sources de données ODBC standard, comme la base de données Microsoft Access. Cependant, Sun ne recommande pas l'utilisation de ce pont pour autre chose que du développement et un déploiement très limité. En particulier, les développeurs de servlets devraient suivre cet avertissement car tout problème dans la portion de code natif du pilote JDBC-ODBC peut planter le serveur en entier, pas uniquement vos servlets.

Les pilotes JDBC sont disponibles pour la majorité des bases de données, chez différents fournisseurs et dans différentes versions. Il existe quatre catégories de pilotes :

Type 1 : pont JDBC-ODBC

Les pilotes de Type 1 utilisent une technologie de pont pour connecter un client Java à un service de base de données ODBC. Le pont JDBC-ODBC de Sun est le pilote de Type 1 le plus répandu. Ces pilotes sont implémentés avec du code natif.

Type 2 : API native et un peu de Java

Les pilotes de Type 2 posent une fine couche de Java autour de bibliothèques en code natif propres à une base de données. Pour les bases de données Oracle, les bibliothèques de code natif peuvent être basées sur les bibliothèques OCI (*Oracle Call Interface*) conçues à l'origine pour des programmeurs C/C++. Les pilotes de Type 2 étant implémentés en utilisant du code natif, dans certains cas ils ont des performances supérieures à leurs équivalents écrits complètement en Java. Cependant, ils ajoutent un facteur de risque car un défaut dans le code natif du pilote peut planter le serveur en entier.

Type 3 : protocole réseau tout en Java

Les pilotes de Type 3 communiquent *via* un protocole réseau générique avec un petit middleware personnalisé. Le composant middleware utilise n'importe quel type de pilote pour fournir l'accès réel à la base de données. Ces pilotes sont en Java, ce qui les rend utiles pour le déploiement d'applets et sûrs pour le déploiement de servlets.

Type 4 : protocole natif tout en Java

Les pilotes de Type 4 sont les plus directs du lot. Écrits entièrement en Java, ils comprennent les protocoles réseau spécifiques à la base de données et peuvent y accéder directement sans autre logiciel.

Une liste des pilotes JDBC disponibles se trouve à *http://industry.java.sun.com/products/jdbc/drivers*.

Obtenir une connexion

La première étape dans l'utilisation d'un pilote JDBC pour obtenir une connexion à la base de données consiste à charger la classe du pilote spécifique dans la JVM de l'application. Le pilote est ainsi disponible pour plus tard, lorsque nous en aurons besoin pour ouvrir la connexion. Le plus facile de charger le pilote consiste à utiliser la méthode `Class.forName()` :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Lorsque le pilote est chargé en mémoire, il s'enregistre tout seul auprès de la classe `java.sql.DriverManager` afin d'être disponible.

L'étape suivante est de demander à la classe `DriverManager` d'ouvrir une connexion à la base de données précisée sous forme d'une URL dont le format est spécifique. La méthode utilisée pour ouvrir la connexion est `DriverManager.getConnection()`. Elle retourne une classe implémentant l'interface `java.sql.Connection` :

```
Connection con =
  DriverManager.getConnection("jdbc:odbc:somedb", "user", "passwd");
```

Une URL JDBC identifie une base de données individuelle d'une façon spécifique au pilote. Différents pilotes peuvent nécessiter différentes informations dans l'URL pour spécifier la base de données hôte. Les URL JDBC commencent habituellement par `jdbc:`*`sous-protocole`*`:`*`sous-nom`*. Par exemple, le pilote Oracle JDBC-Thin utilise une URL de la forme `jdbc:oracle:thin:@`*`hôte-bd`*`:`*`port`*`:`*`sid`* ; le pont JDBC-ODBC utilise `jdbc:odbc:`*`nom-source-de-données`*`;`*`options-odbc`*.

Pendant l'appel à `getConnection()`, l'objet `DriverManager` demande à chaque pilote enregistré s'il reconnaît l'URL. Si un pilote dit oui, le gestionnaire de pilotes l'utilise pour créer l'objet `Connection`. Voici un bout de code qu'une servlet peut utiliser pour charger le pilote de sa base de données avec le pont JDBC-ODBC et créer une connexion initiale :

```
Connection con = null;
try {
  // Charger (et donc enregistrer) le pont JDBC-ODBC.
  // Peut lancer une ClassNotFoundException.
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    // Obtenir une connexion à la base de données.
    // Peut lancer une SQLException.
    con = DriverManager.getConnection("jdbc:odbc:somedb", "user", "passwd");

    //  Le reste du code va ici.

  }
  catch (ClassNotFoundException e) {
    // Gérer une erreur de chargement du pilote.
  }
  catch (SQLException e) {
    // Gérer une erreur d'obtention de la connexion.
  }
  finally {
    // Fermer la connexion pour libérer les ressources de la base de données.
    try {
      if (con != null) con.close();
    }
    catch (SQLException ignored) { }
  }
```

`getConnection()` existe en fait sous trois formes. En plus de celle montrée précédemment, il en existe une plus simple qui prend juste une URL : `getConnection(String url)`. Cette méthode peut être utilisée quand il n'y a pas de contraintes sur l'ouverture de session ou quand les informations d'ouverture de session doivent être placées dans l'URL. Il existe aussi une autre forme prenant une URL et un objet `Properties` : `getConnection(String url, Properties props)`. Cette méthode apporte plus de souplesse. L'objet `Properties` (un `Hashtable` dont toutes les clés et les valeurs sont de type `String`) contient les propriétés `user` et `password` standard ainsi que d'autres propriétés supplémentaires dont a besoin le pilote de base de données sous-jacent. Par exemple, certains pilotes respectent la propriété `cacherows` qui spécifie le nombre de lignes devant être placées dans un cache. L'emploi de cette méthode facilite l'établissement d'une connexion basée sur un fichier *.properties* externe.

Par exemple, le pilote, l'URL et les certificats à utiliser peuvent être spécifiés dans le fichier *sql.properties* suivant. Ce format `nom=valeur` est standard pour les fichiers de propriétés Java :

```
connection.driver=sun.jdbc.odbc.JdbcOdbcDriver
connection.url=jdbc:odbc:somedb
user=user
password=passwd
```

Le code montré à l'*exemple 9-1* établit une connexion à une base de données à partir des valeurs stockées dans ce fichier *sql.properties*. Notez que les propriétés `user` et `password` sont standard, les propriétés `connection.driver` et `connection.url` sont des noms spéciaux utilisés par le code, et toute propriété supplémentaire est aussi passée au pilote sous-jacent.

Exemple 9-1. Utiliser un fichier de propriétés pour ouvrir une connexion à une base de données

```
// Obtenir les propriétés pour la connexion à la base de données.
Properties props = new Properties();
InputStream in = new FileInputStream("sql.properties");
props.load(in);
in.close();  // Devrait se trouver dans un bloc finally.

// Charger le pilote.
Class.forName(props.getProperty("connection.driver"));

// Obtenir la connexion.
con = DriverManager.getConnection(
        props.getProperty("connection.url"), props);
```

Tout d'abord, l'objet `Properties` est construit et rempli avec les valeurs lues à partir du fichier *sql.properties*. Ensuite, les propriétés sont utilisées pour établir la connexion à la base de données. En suivant cette approche, toutes les informations de connexion à la base de données peuvent être modifiées sans avoir à recompiler le code Java !

Obtenir une connexion à partir d'une servlet

Une servlet peut utiliser la même approche pour charger les informations de connexion à une base de données à partir d'un fichier de propriétés stocké dans le répertoire *WEB-INF* de l'application web. La méthode `getResourceAsStream()` permet de retrouver le contenu du fichier :

```
Properties props = new Properties();
InputStream in = getServletContext().getResourceAsStream(
                                        "/WEB-INF/sql.properties");
props.load(in);
in.close();
```

Cependant, les servlets étant souvent déployées à l'aide d'outils graphiques, nous pouvons avancer d'un cran et utiliser ces outils pour configurer la base de données. Une manière de faire consiste à placer des connexions dans un serveur JNDI où une servlet peut localiser une connexion par son nom. Cette approche est généralement employée par des servlets conçues pour s'exécuter dans des serveurs J2EE. Le chapitre 12, *Servlets d'entreprise et J2EE*, présente cela en détail. Une autre approche exploitant les outils de déploiement graphiques mais ne nécessitant pas un serveur JNDI est d'utiliser les paramètres d'initialisation du contexte pour y mettre les informations de configuration. Ces paramètres d'initialisation peuvent être configurés à l'aide des outils de déploiement graphiques, puis sauvés dans le fichier *web.xml* pour faciliter un déploiement portable. (Pour plus d'informations sur les paramètres d'initialisation du contexte, reportez-vous au chapitre 4, *Retrouver l'information*.)

La classe `ContextProperties` montrée à l'*exemple 9-2* rend les paramètres d'initialisation du contexte disponibles dans un objet `Properties`. Cela permet de passer toutes les paires nom/valeur des paramètres d'initialisation à la méthode `DriverManager.getConnection()`.

Exemple 9-2. La classe ContextProperties

```
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ContextProperties extends Properties {

  public ContextProperties(ServletContext context) {
    Enumeration props = context.getInitParameterNames();
    while (props.hasMoreElements()) {
      String name = (String) props.nextElement();
      String value = (String) context.getInitParameter(name);
      put(name, value);
    }
  }
}
```

Une servlet utilise cette classe à la place de charger les données à partir du fichier *sql.properties*, comme le montre le code suivant :

```
// Obtenir les paramètres d'initialisation du contexte sous forme d'un
// objet Properties.
ContextProperties props = new ContextProperties(getServletContext());

// Charger le pilote.
Class.forName(props.getProperty("connection.driver"));

// etc...
```

Il vous suffit de donner aux paramètres d'initialisation du contexte les bonnes valeurs pour votre base de données, soit manuellement dans le fichier *web.xml*, soit en utilisant un outil de déploiement, et tout est prêt à fonctionner.

Exécuter des requêtes SQL

Pour réellement utiliser une base de données, il nous faut un moyen d'exécuter des requêtes. La façon la plus simple d'exécuter une requête est d'utiliser la classe `java.sql.Statement`. Les objets `Statement` ne sont jamais instanciés directement. Un programme appelle la méthode `createStatement()` de `Connection` pour obtenir un nouvel objet `Statement` :

```
Statement stmt = con.createStatement();
```

Une requête qui retourne des données peut être effectuée avec la méthode `executeQuery()` de `Statement`. Cette méthode exécute l'instruction et retourne un `java.sql.ResultSet` qui encapsule les données retrouvées :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
```

Vous pouvez voir un objet `ResultSet` comme une représentation du résultat de la requête retourné une ligne à la fois. Vous utilisez la méthode `next()` de `ResultSet` pour vous déplacer de ligne en ligne. L'interface `ResultSet` possède également une multitu-

de de méthodes conçues pour retrouver les données à partir de la ligne courante. Les méthodes `getString()` et `getObject()` sont parmi celles les plus fréquemment utilisées pour retrouver les valeurs de colonnes :

```
while(rs.next()) {
  String event = rs.getString("event");
  Object count = (Integer) rs.getObject("count");
}
```

Il faut que vous sachiez que le `ResultSet` est lié à son `Statement` parent. Ainsi, si un `Statement` est fermé ou utilisé pour exécuter une autre requête, tous les objets `ResultSet` liés sont fermés automatiquement.

L'*exemple 9-3* montre une servlet très simple qui utilise le pilote JDBC d'Oracle pour effectuer une requête simple, afficher les noms et les numéros de téléphone de tous les employés présents dans une table. Nous supposons que la base contient une table appelée `EMPLOYEES`, avec au moins les deux champs `NAME` et `PHONE`.

Exemple 9-3. Une servlet avec JDBC

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DBPhoneLookup extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    try {
      // Charger (et donc enregistrer) le pilote Oracle.
      Class.forName("oracle.jdbc.driver.OracleDriver");

      // Obtenir une connexion à la base de données.
      con = DriverManager.getConnection(
        "jdbc:oracle:thin:@dbhost:1528:ORCL", "user", "passwd");

      // Créer un objet Statement.
      stmt = con.createStatement();

      // Exécuter une requête SQL, obtenir un ResultSet.
      rs = stmt.executeQuery("SELECT NAME, PHONE FROM EMPLOYEES");

      // Afficher le résultat sous forme d'une liste.
```

Exemple 9-3. Une servlet avec JDBC (suite)

```
      out.println("<HTML><HEAD><TITLE>Répertoire téléphonique" +
                  "</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<UL>");
      while(rs.next()) {
        out.println("<LI>" + rs.getString("name") + " " +
                    rs.getString("phone"));
      }
      out.println("</UL>");
      out.println("</BODY></HTML>");
    }
    catch(ClassNotFoundException e) {
      out.println("Impossible de charger le pilote de base de données : " +
                  e.getMessage());
    }
    catch(SQLException e) {
      out.println("SQLException interceptée : " + e.getMessage());
    }
    finally {
      // Toujours fermer la connexion à la base de données.
      try {
        if (con != null) con.close();
      }
      catch (SQLException ignored) { }
    }
  }
}
```

C'est peut-être la servlet pour base de données la plus simple qu'il vous soit possible de voir. DBPhoneLookup ne fait que se connecter à la base de données, effectuer une requête qui retrouve les noms et numéros de téléphone de tous ceux qui se trouvent dans la table EMPLOYEES et afficher cette liste à l'utilisateur.

Gérer les exceptions SQL

La plus grande partie du code de DBPhoneLookup se trouve dans un bloc try/catch. Ce bloc intercepte deux exceptions : ClassNotFoundException et SQLException. La première est lancée par la méthode Class.forName() lorsque la classe du pilote JDBC ne peut être chargée. La seconde est lancée par toute méthode JDBC ayant un problème. Les objets SQLException sont comme les autres types d'exception, sauf qu'ils peuvent être chaînés. La classe SQLException définit une méthode additionnelle, getNextException(), qui permet à l'exception d'encapsuler d'autres objets Exception. Nous ne nous sommes pas occupés de cette fonctionnalité dans l'exemple précédent, mais voici comment l'utiliser :

```
catch (SQLException e) {
  out.println(e.getMessage());
  while((e = e.getNextException()) != null) {
```

```
        out.println(e.getMessage());
      }
    }
```

Ce code affiche le message de la première exception et ensuite boucle sur toutes les exceptions restantes, affichant le message d'erreur qui leur est associé. En pratique, la première exception inclut généralement l'information la plus pertinente.

Détail des ensembles de résultats

Avant de continuer, nous devons mieux examiner l'interface `ResultSet` et l'interface liée `ResultSetMetaData`. Dans l'*exemple 9-3*, nous savions à quoi ressemblait notre requête, et sachant ce que nous allions recevoir, nous avons donc correctement formaté la sortie. Mais, si nous voulons afficher les résultats d'une requête dans une table HTML, il serait bien d'avoir du code Java qui construit la table automatiquement à partir du `ResultSet` au lieu de toujours réécrire le même code d'affichage itératif. De plus, ce genre de code permet de changer le contenu de la table simplement en changeant la requête.

L'interface `ResultSetMetaData` fournit à un programme un moyen de connaître à la volée la structure du résultat d'une requête. Nous pouvons l'utiliser pour construire un objet qui génère dynamiquement une table HTML à partir d'un `ResultSet`, comme montré à l'*exemple 9-4*. Plusieurs outils Java de création de contenu HTML ont des possibilités similaires (voir les chapitres 14 à 18).

Exemple 9-4. Une classe pour générer une table HTML à partir d'un ResultSet à l'aide de ResultSetMetaData

```
import java.sql.*;

public class HtmlResultSet {

  private ResultSet rs;

  public HtmlResultSet(ResultSet rs) {
    this.rs = rs;
  }

  public String toString() {  // Peut être appelée au plus une fois.
    StringBuffer out = new StringBuffer();
    // Commencer une table pour afficher le résultat.
    out.append("<TABLE>\n");

    try {
      ResultSetMetaData rsmd = rs.getMetaData();

      int numcols = rsmd.getColumnCount();

      // Intituler la table avec les étiquettes de colonne du résultat.
      out.append("<TR>");
      for (int i = 1; i <= numcols; i++) {
        out.append("<TH>" + rsmd.getColumnLabel(i));
      }
```

Exemple 9-4. Une classe pour générer une table HTML à partir d'un ResultSet à l'aide de ResultSetMetaData (suite)

```
      out.append("</TR>\n");

      while(rs.next()) {
        out.append("<TR>"); // Comencer une nouvelle ligne.
        for (int i = 1; i <= numcols; i++) {
          out.append("<TD>"); // Commencer un nouvel élément de donnée.
          Object obj = rs.getObject(i);
          if (obj != null)
            out.append(obj.toString());
          else
            out.append(" ");
        }
        out.append("</TR>\n");
      }

      // Terminer la table.
      out.append("</TABLE>\n");
    }
    catch (SQLException e) {
      out.append("</TABLE><H1>ERREUR :</H1> " + e.getMessage() + "\n");
    }

    return out.toString();
  }
}
```

Cet exemple montre comment utiliser deux méthodes de base de `ResultSetMetaData` : `getColumnCount()` et `getColumnLabel()`. La première retourne le nombre de colonnes dans `ResultSet`, la seconde retrouve le nom d'une colonne particulière dans le résultat en fonction de son indice numérique. Les indices des objets `ResultSet` suivent le standard des SGBDR au lieu du standard de C++/Java, ce qui signifie qu'ils vont de 1 à *n* au lieu de 0 à *n*-1.

Cet exemple utilise aussi la méthode `getObject()` de `ResultSet` pour retrouver la valeur de chaque colonne. Toutes les méthodes `getXXX()` fonctionnent avec des indices de colonnes et des noms de colonnes. L'accès aux données de cette façon est plus efficace et, avec du SQL bien écrit, plus portable. Nous utilisons ici `getObject().toString()` au lieu de `getString()` afin de simplifier la gestion des valeurs `null`, comme expliqué dans la prochaine section.

Le *tableau 9-1* montre les méthodes Java que vous pouvez utiliser pour retrouver certains types de donnée SQL communs à partir d'une base de données. Quel que soit le type, vous pouvez toujours utiliser la méthode `getObject()` de `ResultSet`, auquel cas le type de l'objet retourné est montré dans la seconde colonne. Vous pouvez aussi utiliser la méthode `getXXX()` spécifique. Ces méthodes sont montrées dans la troisième colonne avec les types de données Java qu'elles retournent. Souvenez-vous que les types de données SQL supportés varient d'une base de données à une autre.

Tableau 9-1. Méthodes pour retrouver les données d'un ResultSet

Type de données SQL	Type Java retourné par getObject()	Alternative à getObject() recommandée
BIGINT	Long	long getLong()
BINARY	byte[]	byte[] getBytes()
BIT	Boolean	boolean getBoolean()
CHAR	String	String getString()
DATE	java.sql.Date	java.sql.Date getDate()
DECIMAL	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
DOUBLE	Double	double getDouble()
FLOAT	Double	double getDouble()
INTEGER	Integer	int getInt()
LONGVARBINARY	byte[]	InputStream getBinaryStream()
LONGVARCHAR	String	InputStream getAsciiStream() InputStream getUnicodeStream()
NUMERIC	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
REAL	Float	float getFloat()
SMALLINT	Integer	short getShort()
TIME	Java.sql.Time	java.sql.Time getTime()
TIMESTAMP	Java.sql.Timestamp	java.sql.Timestamp getTimestamp()
TINYINT	Integer	byte getByte()
VARBINARY	byte[]	byte[] getBytes()
VARCHAR	String	String getString()

Gérer les champs null

Un champ d'une base de données peut être mis à null pour indiquer qu'aucune valeur n'est présente, exactement comme un objet Java peut être mis à null. La gestion des valeurs null d'une base de données avec JDBC peut être un peu compliquée. Si la méthode ne retourne pas un objet, comme getInt(), elle ne peut indiquer si la colonne est null ou si elle contient une information. Toute valeur spéciale comme 0 peut être une valeur valide. C'est pourquoi JDBC inclut la méthode wasNull() dans ResultSet. Elle retourne true si la dernière lecture d'une colonne était un vrai null de base de données, sinon false. Cela signifie que vous devez lire les données de ResultSet dans une variable, appeler wasNull() et continuer de façon appropriée. Ce n'est pas très joli mais cela fonctionne. Voici un exemple :

```
int age = rs.getInt("age");
if (!rs.wasNull())
  out.println("Age : " + age);
```

Une autre façon de détecter les valeurs null est d'utiliser la méthode getObject(). Si une colonne est null, getObject() retourne toujours null. L'utilisation de getObject() supprime l'obligation d'appeler wasNull() et conduit à un code plus simple, mais les objets retournés ne sont pas aussi faciles à utiliser que les types primitifs.

Mettre à jour la base de données

La plupart des sites web connectés à une base de données veulent faire plus qu'effectuer des requêtes. Lorsqu'un client soumet une commande ou fournit des informations, les données doivent être entrées dans la base de données. Lorsque vous savez que vous êtes en train d'exécuter une instruction SQL UPDATE, INSERT ou DELETE et que vous savez que vous n'attendez pas un ResultSet, vous pouvez utiliser la méthode executeUpdate() de Statement. Elle retourne une valeur indiquant le nombre de lignes modifiées par l'instruction. Voici comment l'utiliser :

```
int count =
  stmt.executeUpdate("DELETE FROM CUSTOMERS WHERE CUSTOMER_ID = 5");
```

Si vous exécutez une instruction SQL qui peut retourner soit un ResultSet ou un décompte (par exemple, si vous traitez du SQL soumis par l'utilisateur ou si vous construisez des classes génériques de traitement de données), utilisez la méthode générique execute() de Statement. Elle retourne un booléen qui vaut true si l'instruction SQL a produit un objet ResultSet ou plusieurs, et false si le résultat est un compteur de mise à jour :

```
boolean b = stmt.execute(sql);
```

Les méthodes getResultSet() et getUpdateCount() de Statement fournissent l'accès aux résultats de la méthode execute(). L'*exemple 9-5* montre l'utilisation de ces méthodes dans une nouvelle version de HtmlResultSet, appelée HtmlSQLResult, qui crée une table HTML à partir de n'importe quelle instruction SQL.

Exemple 9-5. Une classe pour générer une table HTML à partir d'un ResultSet à l'aide de ResultSetMetaData

```
import java.sql.*;

public class HtmlSQLResult {
  private String sql;
  private Connection con;

  public HtmlSQLResult(String sql, Connection con) {
    this.sql = sql;
    this.con = con;
  }

  public String toString() {  // Peut être appelée au plus une fois.
    StringBuffer out = new StringBuffer();

    // Ôter le commentaire de la ligne suivante pour afficher la commande
    // SQL au début de la table.
    // out.append("Résultat de l'instruction SQL : " + sql + "<P>\n");

    try {
      Statement stmt = con.createStatement();

      if (stmt.execute(sql)) {
```

Exemple 9-5. Une classe pour générer une table HTML à partir d'un ResultSet à l'aide de ResultSetMetaData (suite)

```
        // Il y a un ResultSet à prendre.
        ResultSet rs = stmt.getResultSet();
        out.append("<TABLE>\n");

        ResultSetMetaData rsmd = rs.getMetaData();

        int numcols = rsmd.getColumnCount();

        // Intituler la table avec les étiquettes de colonne du résultat.
        out.append("<TR>");
        for (int i = 1; i <= numcols; i++)
          out.append("<TH>" + rsmd.getColumnLabel(i));
        out.append("</TR>\n");

        while(rs.next()) {
          out.append("<TR>");  // Commencer une nouvelle ligne.
          for(int i = 1; i <= numcols; i++) {
            out.append("<TD>");  // Commencer un nouvel élément de donnée.
            Object obj = rs.getObject(i);
            if (obj != null)
              out.append(obj.toString());
            else
              out.append(" ");
            }
          out.append("</TR>\n");
        }

        // Terminer la table.
        out.append("</TABLE>\n");
      }
      else {
        // Il y a un compteur à prendre.
        out.append("<B>Enregistrements affectés :</B> " +
                   stmt.getUpdateCount());
      }
    }
    catch (SQLException e) {
      out.append("</TABLE><H1>ERREUR :</H1> " + e.getMessage());
    }

    return out.toString();
  }
}
```

Cet exemple utilise `execute()` pour exécuter toute instruction SQL passée au constructeur de `HtmlSQLResult`. Ensuite, en fonction de la valeur retournée, on appelle soit `getResultSet()` soit `getUpdateCount()`. Notez que ni `getResultSet()` ni `getUpdateCount()` ne doivent être appelés plus d'une fois par requête.

Utiliser des instructions préparées

Un objet PreparedStatement est comme un objet Statement normal, en cela qu'il peut être utilisé pour exécuter des instructions SQL. La différence importante est que le SQL dans un PreparedStatement est précompilé par la base de données pour une exécution plus rapide. Une fois un PreparedStatement compilé, il peut être encore personnalisé en ajustant des paramètres prédéfinis. Les instructions préparées sont utiles dans des applications qui doivent exécuter plusieurs fois la même commande SQL.

Utilisez la méthode prepareStatement(String) de Connection pour créer des objets PreparedStatement. Le caractère ? représente un emplacement pour des valeurs données plus tard. Par exemple :

```
PreparedStatement pstmt = con.prepareStatement(
  "INSERT INTO ORDERS (ORDER_ID, CUSTOMER_ID, TOTAL) VALUES (?,?,?)");

// Autre code.

pstmt.clearParameters();    // Effacer toute valeur précédente
                            // des paramètres.
pstmt.setInt(1, 2);         // Définir ORDER_ID.
pstmt.setInt(2, 4);         // Définir CUSTOMER_ID.
pstmt.setDouble(3, 53.43); // Définir TOTAL.
pstmt.executeUpdate();      // Exécuter le SQL stocké.
```

La méthode clearParameters() supprime toutes les valeurs de paramètre définies précédemment, tandis que les méthodes set*XXX*() sont utilisées pour affecter les valeurs à chaque emplacement représenté par un point d'interrogation. Une fois les valeurs assignées à tous les paramètres, appelez executeUpdate() pour exécuter le PreparedStatement.

La classe PreparedStatement a une application importante avec les servlets. Lors du chargement d'un texte soumis par l'utilisateur dans une base de données en utilisant des objets Statement et du SQL dynamique, vous devez faire attention à ne pas introduire accidentellement des caractères de contrôle SQL (comme " ou ') sans leur appliquer un échappement comme le veut votre base de données. Avec une base de données comme Oracle qui entoure les chaînes de caractères par des apostrophes, si vous essayez d'insérer "John d'Artagan" dans la base, vous obtiendrez ce SQL corrompu :

```
INSERT INTO MUSKETEERS (NAME) VALUES ('John d'Artagan')
```

Comme vous pouvez le voir, la chaîne se termine deux fois. Une solution est de remplacer manuellement l'apostrophe (') par deux apostrophes (''), la séquence d'échappement d'Oracle pour une apostrophe. Cette solution vous oblige à appliquer l'échappement à chaque caractère traité de façon spéciale par votre base de données ; c'est une tâche difficile et incohérente avec l'écriture de code indépendant de la plateforme. Une bien meilleure solution consiste à utiliser un PreparedStatement et à lui passer la chaîne en utilisant sa méthode setString(), comme montré ci-après. PreparedStatement applique automatiquement l'échappement à la chaîne conformément à votre base de données :

```
PreparedStatement pstmt = con.prepareStatement(
  "INSERT INTO MUSKETEERS (NAME) VALUES (?)");
```

```
pstmt.setString(1, "John d'Artagan");
pstmt.executeUpdate();
```

Réutiliser des objets de base de données

Dans l'introduction, nous avons mentionné que le cycle de vie d'une servlet permettait des accès à la base de données extrêmement rapides. Après avoir utilisé JDBC quelque temps, il devient évident que le principal goulet d'étranglement pour les performances vient souvent dès le début, lorsque l'on ouvre une connexion à une base de données. C'est rarement un problème pour la plupart des applications et les applets car elles peuvent s'offrir le luxe d'attendre quelques secondes pour créer le `Connection` qui sera utilisé pendant toute la durée de vie du programme. Avec les servlets, ce goulet est plus sérieux car nous créons et détruisons un `Connection` à chaque requête. Heureusement, le cycle de vie d'une servlet nous permet de réutiliser la même connexion pour de multiples requêtes, même des requêtes concurrentes car les objets `Connection` sont sûrs vis-à-vis des threads.

Réutiliser des connexions à une base de données

Une servlet peut créer plusieurs objets `Connection` dans sa méthode `init()` et les réutiliser dans ses méthodes `service()`, `doGet()` et `doPost()`. L'*exemple 9-6* montre la servlet d'annuaire réécrite afin de créer par avance son objet `Connection`. Elle utilise aussi la classe `HtmlSQLResult` de l'*exemple 9-5* pour afficher les résultats. Notez que cette servlet utilise le pilote JDBC de Sybase.

Exemple 9-6. Une servlet d'annuaire améliorée

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DBPhoneLookupReuse extends HttpServlet {

  private Connection con = null;

  public void init() throws ServletException {
    try {
      // Charger (et donc enregistrer) le pilote Sybase.
      Class.forName("com.sybase.jdbc.SybDriver");
      con = DriverManager.getConnection(
        "jdbc:sybase:Tds:dbhost:7678", "user", "passwd");
    }
    catch (ClassNotFoundException e) {
      throw new UnavailableException(
          "Impossible de charger le pilote de base de données");
    }
    catch (SQLException e) {
      throw new UnavailableException(
          "Impossible d'obtenir la connexion à la base de données");
```

Exemple 9-6. Une servlet d'annuaire améliorée (suite)

```
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HTML><HEAD><TITLE>Annuaire téléphonique</TITLE></HEAD>");
    out.println("<BODY>");

    HtmlSQLResult result =
      new HtmlSQLResult("SELECT NAME, PHONE FROM EMPLOYEES", con);

    // Afficher la sortie résultante.
    out.println("<H2>Employés :</H2>");
    out.println(result);
    out.println("</BODY></HTML>");
  }

  public void destroy() {
    // Libération des ressources.
    try {
      if (con != null) con.close();
    }
    catch (SQLException ignored) { }
  }
}
```

Réutiliser des instructions préparées

Vous pouvez encore améliorer sans trop d'efforts les performances des servlets en créant par avance d'autres objets pour la base de données. L'objet `PreparedStatement` est un candidat idéal car il peut précompiler une instruction SQL. Cela ne fait gagner en général que quelques millisecondes, mais si votre site reçoit des centaines de milliers de hits pas jour, cela peut monter très vite.

Notez cependant que le partage d'objets autres que les connexions pose un problème. Les servlets doivent être sûres vis-à-vis des threads, et accéder à un `PreparedStatement` peut demander trois ou quatre appels de méthodes. Si un thread appelle la méthode `clearParameters()` de `PreparedStatement` juste avant qu'un autre thread appelle `execute()`, les résultats de `execute()` seront désastreux. Il y a aussi la limitation de `Statement` qui n'accepte qu'une seule requête (et les ensembles de résultats associés) à la fois. Une solution est de synchroniser les sections de votre code qui utilisent les objets partagés, comme l'explique le chapitre 3 et montré ici :

```
synchronized (pstmt) {
  pstmt.clearParameters();
  pstmt.setInt(1, 2);
```

```
    pstmt.setInt(2, 4);
    pstmt.setDouble(3, 53.43);
    pstmt.executeUpdate();
  }
```

Malheureusement cette solution apporte son lot d'inconvénients. Sur certaines plates-formes, entrer dans un bloc de synchronisation prend du temps et les objets synchronisés ne peuvent être utilisés que par un seul thread à la fois. Cependant, certaines servlets ont déjà besoin d'un bloc synchronisé et, dans ce cas, ces inconvénients sont moindres. Une bonne règle est de créer vos connexions par avance ainsi que les objets fréquemment utilisés (comme les objets `PreparedStatement`) qui peuvent être rapidement utilisés dans des blocs de synchronisation pré-existants.

Pour les servlets écrites avec l'interface `SingleThreadModel`, ces problèmes ne se posent pas. D'un autre côté, vous aurez de multiples copies de votre servlet chargées à la fois, ce qui peut être au détriment des performances.

Transactions

Jusqu'ici nous n'avons pas mentionné une fonctionnalité importante des systèmes de bases de données modernes : les transactions. La plupart des sites web orientés services ont besoin d'exécuter une ou plusieurs instructions `SELECT` et d'insérer des morceaux de données. Examinons une application bancaire en ligne. Pour effectuer le transfert de 10 000 € entre deux comptes, votre programme doit effectuer une opération qui consiste en deux actions séparées mais liées : créditer un compte et en débiter un autre. Maintenant, imaginez que pour une raison ou pour une autre l'instruction SQL du crédit réussisse et que celle du débit échoue. Un compte sera plus riche de 10 000 €, mais l'autre compte n'aura pas été débité.

Un échec SQL n'est pas le seul problème potentiel. Si un autre utilisateur vérifie l'équilibre des comptes entre le crédit et le débit, il verra l'équilibre originel. La base de données est vue dans un état invalide (le montant présenté est plus important qu'il ne l'est en réalité). D'accord, ce genre de choses a peu de chance d'arriver souvent, mais dans un univers de possibilités infinies, il est certain que cela arrivera. Ce genre de problème est similaire aux problèmes de synchronisation montrés au chapitre 3. Cette fois-ci, au lieu de nous préoccuper de la validité des données stockées dans la servlet, nous sommes concernés par la validité de la base de données sous-jacente. Une simple synchronisation n'est pas suffisante pour résoudre ce problème : de multiples servlets pourraient accéder à la même base de données. Pour des systèmes comme les logiciels bancaires, il y a de bonnes chances que la base de données soit aussi utilisée par de nombreuses applications non entièrement écrites en Java.

Voilà un problème plutôt complexe, non ? Heureusement, il existait bien avant que Java n'arrive, il a donc déjà été résolu. La plupart des SGBDR importants supportent le concept de *transaction*. Une transaction vous permet de regrouper plusieurs instructions SQL. En utilisant ces SGBDR, vous pouvez commencer une transaction, effectuer toutes les actions que vous voulez et, soit valider (*commit*) les résultats, soit défaire (*roll-back*) toutes vos instructions SQL. Si nous construisons notre application bancaire en ligne avec un système basé sur les transactions, le crédit sera automatiquement annulé si le débit échoue.

Une transaction est isolée du reste de la base de données jusqu'à ce qu'elle soit terminée. En ce qui concerne la base de données, tout se passe en une fois (en d'autres termes, les transactions sont atomiques). Cela signifie que tous les autres utilisateurs accédant à la base de données auront toujours une vue valide des données, mais peut-être pas une vue à jour. Si un utilisateur demande un rapport sur les articles vendus avant que votre transaction de vente d'un article soit terminée, le rapport n'inclura pas la vente la plus récente.

Utiliser les transactions avec JDBC

La gestion des transactions avec JDBC se fait *via* l'objet `Connection`. Par défaut, les nouvelles connexions commencent en mode d'*auto-validation*. Cela signifie que chaque instruction SQL est exécutée comme une transaction individuelle immédiatement validée dans la base de données. Pour contrôler vous-même la validation, autorisant ainsi le regroupement d'instructions SQL en transactions, appelez `setAutoCommit(false)` sur l'objet `Connection`. Vous pouvez vérifier l'état de l'auto-validation avec la méthode `getAutoCommit()`. Une fois toutes vos instructions SQL terminées, appelez `commit()` pour enregistrer de façon permanente la transaction dans la base de données. Ou, si vous avez rencontré une erreur, appelez `rollback()` pour la défaire.

L'*exemple 9-7* montre une servlet qui utilise les transactions pour un traitement basique des commandes. Elle suppose l'existence de deux tables dans la base ODBC, `INVENTORY` (contenant l'identifiant d'un produit et la quantité en stock) et `SHIPPING` (contenant un identifiant de produit, un numéro de commande et la quantité envoyée). La servlet utilise une méthode `chargeCard()` non montrée pour la facturation et lançant une exception si le numéro de carte du client n'est pas valide.

Exemple 9-7. Traitement des commandes basé sur les transactions

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class OrderHandler extends HttpServlet {

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    Connection con = null;
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      con = DriverManager.getConnection(
                "jdbc:odbc:ordersdb", "user", "passwd");

      // Activer les transactions.
      con.setAutoCommit(false);
```

Exemple 9-7. Traitement des commandes basé sur les transactions (suite)

```
      Statement stmt = con.createStatement();
      stmt.executeUpdate(
        "UPDATE INVENTORY SET STOCK = (STOCK - 10) WHERE PRODUCTID = 7");
      stmt.executeUpdate(
        "UPDATE SHIPPING SET SHIPPED = (SHIPPED + 10) WHERE PRODUCTID = 7");

      chargeCard();  // Méthode qui n'existe pas encore...

      con.commit();
      out.println("Commande réussie ! Merci !");
    }
    catch (Exception e) {
      // Toute erreur conduit à défaire la transaction.
      try {
        con.rollback();
      }
      catch (SQLException ignored) { }
      out.println(
        "Échec de la commande. Veuillez contacter le support technique.");
    }
    finally {
      // Libérer les ressources.
      try {
        if (con != null) con.close();
      }
      catch (SQLException ignored) { }
    }
  }
}
```

Voici quelques remarques à propos de cet exemple. Premièrement, la logique transactionnelle d'une commande est dans `doPost()` puisque l'action du client n'est pas du tout répétable de façon sûre. Deuxièmement, cet exemple démontrant la logique transactionnelle plutôt que celle de la servlet, celle-ci suppose simplement que l'utilisateur achète 10 unités de l'article 7, au lieu de se préoccuper de l'analyse d'un formulaire pour la carte de crédit et l'information sur la commande. Enfin, lorsque la servlet s'exécute, toute exception lancée pendant l'initialisation du pilote, la connexion à la base de données, l'exécution d'une instruction SQL ou le débit de la carte de crédit provoque le saut de l'exécution au bloc `catch()`, où la méthode `rollback()` est appelée, défaisant tout notre travail.

Optimiser le processus transactionnel

Notez que dans l'exemple précédent, l'objet `Connection` a été créé dans la méthode `doPost()`, abandonnant l'amélioration des performances obtenue précédemment dans ce chapitre en déplaçant sa création dans `init()`. Ce choix a été fait car les transactions sont liées aux connexions et les connexions utilisant les transactions ne peuvent donc

pas être partagées. Imaginez ce qui pourrait arriver si une autre invocation de cette servlet appelait la méthode `commit()` alors que notre commande n'avait pas atteint la seconde instruction SQL. Notre table `INVENTORY` aurait un trou de 10 unités !

Donc, comment utilisons-nous les transactions sans avoir à se connecter à la base de données chaque fois qu'une page est demandée ? Voici plusieurs possibilités :

- Synchroniser la méthode `doPost()`. Cela signifie que chaque instance de la servlet ne traite qu'une seule requête à la fois. Cela fonctionne bien pour les sites à faible trafic, mais provoque un ralentissement pour vos utilisateurs car chaque transaction doit être terminée avant que la prochaine puisse commencer. Si vous devez effectuer des mises à jour et des insertions fréquentes dans la base de données, le délai pourrait être inacceptable.
- Laisser les choses comme elles sont, mais créer un nouvel objet `Connection` pour chaque transaction. Si vous devez mettre à jour les données une seule fois parmi une centaine de requêtes, c'est peut-être ce qu'il y a de plus simple.
- Créer une réserve d'objets `Connection` dans la méthode `init()` et les utiliser au fur et à mesure des besoins, comme montré à la *figure 9-3*. C'est probablement la façon la plus efficace de traiter le problème, si c'est bien fait. Cependant, cela peut très vite devenir très compliqué sans le support de classes tierces.
- Implémenter le suivi de session dans la servlet et placer dans l'objet `HttpSession` un `Connection` pour chaque utilisateur. Cela vous permet d'aller plus loin que les autres solutions et d'étendre une transaction sur plusieurs requêtes de pages ou même sur plusieurs servlets.

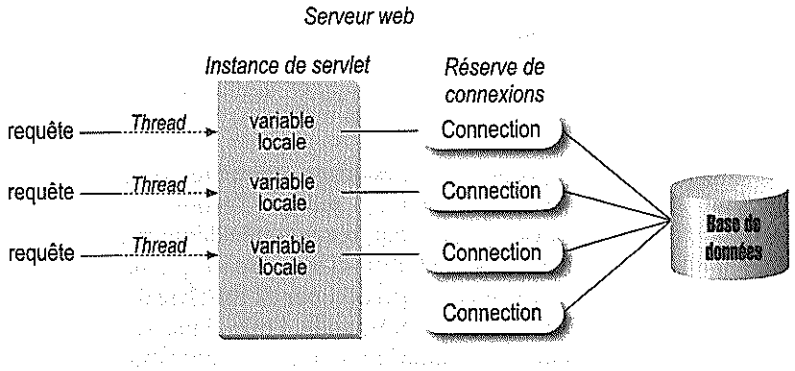


Figure 9-3. Servlets utilisation une réserve de connexions

Réserve de connexions

Pour une servlet complexe, la création d'une réserve de connexions est l'approche idéale. Avec cette réserve, nous pouvons dupliquer uniquement les ressources que nous avons besoin de dupliquer (c'est-à-dire les objets `Connection`). Une réserve de connexions peut aussi gérer intelligemment sa taille et s'assurer que chaque connexion reste valide. Des paquetages de gestion d'une réserve de connexions sont disponibles. Certains, comme `DbConnectionBroker` disponible gratuitement chez Java Exchange à

http://javaexchange.com, fonctionnent en créant un objet qui distribue des connexions et des identifiants de connexion à la demande. D'autres, souvent appelés des *pilotes de réserves*, implémente un nouveau pilote JDBC qui gère une réserve de connexions pour un autre pilote JDBC. L'utilisation de ce genre de pilote facilite l'implémentation de la réserve de connexions dans vos servlets. Cependant, ils ont un coût de fonctionnement un peu plus élevé que les pilotes standards car chaque classe JDBC doit être enveloppée dans une autre classe.

L'*exemple 9-8* montre un système de réserve de connexions simple. Un certain nombre de connexions sont créées au démarrage et sont utilisées en fonction des besoins. Si toutes les connexions sont utilisées, la servlet en crée une nouvelle. Bien que notre classe `ConnectionPool` soit totalement fonctionnelle, un déploiement réel ferait mieux d'utiliser un des paquetages tiers existants bien plus complets.

Exemple 9-8. La classe ConnectionPool

```
import java.sql.*;
import java.util.*;

public class ConnectionPool {
  private Hashtable connections = new Hashtable();
  private Properties props;

  public ConnectionPool(Properties props, int initialConnections)
                  throws SQLException, ClassNotFoundException {
    this.props = props;
    initializePool(props, initialConnections);
  }

  public ConnectionPool(String driverClassName, String dbURL,
                        String user, String password,
                        int initialConnections)
                  throws SQLException, ClassNotFoundException {
    props = new Properties();
    props.put("connection.driver", driverClassName);
    props.put("connection.url", dbURL);
    props.put("user", user);
    props.put("password", password);
    initializePool(props, initialConnections);
  }

  public Connection getConnection() throws SQLException {
    Connection con = null;

    Enumeration cons = connections.keys();

    synchronized (connections) {
      while(cons.hasMoreElements()) {
        con = (Connection)cons.nextElement();

        Boolean b = (Boolean)connections.get(con);
```

Exemple 9-8. La classe ConnectionPool (suite)

```
        if (b == Boolean.FALSE) {
          // Nous avons donc trouvé une connexion inutilisée.
          // Tester son intégrité par un appel à setAutoCommit(true).
          // Pour une utilisation réelle, il faudrait faire plus de tests,
          // comme effectuer une requête simple.
          try {
            con.setAutoCommit(true);
          }
          catch(SQLException e) {
            // Problème avec la connexion, la remplacer.
            connections.remove(con);
            con = getNewConnection();
          }
          // Mettre à jour le Hashtable pour montrer ce qu'on a pris.
          connections.put(con, Boolean.TRUE);
          // Retourner la connexion.
          return con;
        }
      }

      // Si nous arrivons ici c'est qu'il n'y avait plus de connexions
      // libres. Il faut en créer plus. Une réserve de connexions plus
      // robuste aura une limite de taille maximale et récupérera les
      // connexions après un certain temps.
      con = getNewConnection();
      connections.put(con, Boolean.FALSE);
      return con;
    }
  }

  public void returnConnection(Connection returned) {
    if (connections.containsKey(returned)) {
      connections.put(returned, Boolean.FALSE);
    }
  }

  private void initializePool(Properties props, int initialConnections)
                  throws SQLException, ClassNotFoundException {
    // Charger le pilote.
    Class.forName(props.getProperty("connection.driver"));

    // Mettre notre réserve de Connection dans le Hashtable.
    // La valeur FALSE indique qu'elles ne sont pas utilisées.
    for(int i = 0; i < initialConnections; i++) {
      Connection con = getNewConnection();
      connections.put(con, Boolean.FALSE);
    }
  }
```

Exemple 9-8. La classe ConnectionPool (suite)

```
  private Connection getNewConnection() throws SQLException {
    return DriverManager.getConnection(
      props.getProperty("connection.url"), props);
  }
}
```

La classe ConnectionPool gère un Hashtable utilisant des objets Connection comme clés de hachage et des objets Boolean comme valeurs stockées. La valeur Boolean indique si une connexion est utilisée. Un programme appelle la méthode getConnection() de ConnectionPool pour obtenir un objet Connection qu'il peut utiliser ; il appelle returnConnection() pour rendre la connexion à la réserve. Ce modèle de réserve de connexions est plutôt simple. Pour un déploiement réel, vous voudrez probablement quelque chose offrant une meilleure qualité de gestion de la réserve et dont la vérification de l'intégrité est meilleure qu'un simple appel à setAutoCommit().

L'*exemple 9-9* montre une version revue de la servlet de traitement de commandes qui utilise la classe implémentant la réserve de connexions.

Exemple 9-9. Servlet transactionnelle avec réserve de connexions

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class OrderHandlerPool extends HttpServlet {
  private ConnectionPool pool;

  public void init() throws ServletException {
    try {
      pool = new ConnectionPool("oracle.jdbc.driver.OracleDriver",
                                "jdbc:oracle:oci7:orders", "user", "passwd", 5);
    }
    catch (Exception e) {
      throw new UnavailableException(
          "Impossible de créer une réserve de connexions");
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                                  throws ServletException, IOException {
    Connection con = null;

    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    try {
      con = pool.getConnection();
```

Exemple 9-9. Servlet transactionnelle avec réserve de connexions (suite)

```
      // Activer les transactions.
      con.setAutoCommit(false);

      Statement stmt = con.createStatement();
      stmt.executeUpdate(
        "UPDATE INVENTORY SET STOCK = (STOCK - 10) WHERE PRODUCTID = 7");
      stmt.executeUpdate(
        "UPDATE SHIPPING SET SHIPPED = (SHIPPED + 10) WHERE PRODUCTID = 7");

      chargeCard();  // Méthode qui n'existe pas encore...

      con.commit();
      out.println("Commande réussie ! Merci !");
    }
    catch (Exception e) {
      // Toute erreur conduit à défaire la transaction.
      try {
        con.rollback();
      }
      catch (Exception ignored) { }
      out.println(
        "Échec de la commande. Veuillez contacter le support technique.");
    }
    finally {
      if (con != null) pool.returnConnection(con);
    }
  }
}
```

Connexions faisant partie de la session

Le suivi de session que nous avons vu en détail au chapitre 7, *Suivi de session*, nous donne un autre moyen de gérer les transactions. En utilisant les sessions, nous pouvons créer ou allouer une connexion à une base de données dédiée à chaque utilisateur d'un site web ou d'une application intranet. L'*exemple 9-10* montre cela avec une servlet `ConnectionPerClient` qui associe un objet `Connection` unique à chaque `HttpSession` du client. Elle enveloppe l'objet `Connection` avec un `ConnectionHolder` responsable de la gestion du cycle de vie de la connexion.

Exemple 9-10. Associer une connexion à chaque session

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

class ConnectionHolder implements HttpSessionBindingListener {
  private Connection con = null;
```

Exemple 9-10. Associer une connexion à chaque session (suite)

```
  public ConnectionHolder(Connection con) {
    // Sauver le Connection.
    this.con = con;
    try {
      con.setAutoCommit(false);  // Les transactions peuvent s'étendre sur
    }                            // plusieurs pages web !
    catch(SQLException e) {
      // Effectuer la gestion des erreurs.
    }
  }

  public Connection getConnection() {
    return con;  // Retourner le cargo.
  }

  public void valueBound(HttpSessionBindingEvent event) {
    // Ne rien faire quand ajouté à une Session.
  }

  public void valueUnbound(HttpSessionBindingEvent event) {
    // Défaire les modifications quand supprimé d'une session
    // (ou quand la session expire).
    try {
      if (con != null) {
        con.rollback();  // Oublier les données non validées.
        con.close();
      }
    }
    catch (SQLException e) {
      // L'indiquer.
    }
  }
}

/* La vraie servlet. */

public class ConnectionPerClient extends HttpServlet {

  public void init() throws ServletException {
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
    }
    catch (ClassNotFoundException e) {
      throw new UnavailableException("Impossible de charger OracleDriver");
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
```

Exemple 9-10. Associer une connexion à chaque session (suite)

```
res.setContentType("text/plain");
PrintWriter out = res.getWriter();

HttpSession session = req.getSession(true);
Connection con;

// Synchronisation : sans cela, deux détenteurs peuvent être créés
// pour un client.
synchronized (session) {
  // Essayer d'obtenir le détenteur de la connexion pour ce client.
  ConnectionHolder holder =
    (ConnectionHolder) session.getAttribute("servletapp.connection");

  // Créer (et stocker) une nouvelle connexion et un détenteur
  // si nécessaire.
  if (holder == null) {
    try {
      holder = new ConnectionHolder(DriverManager.getConnection(
        "jdbc:oracle:oci7:ordersdb", "user", "passwd"));
      session.setAttribute("servletapp.connection", holder);
    }
    catch (SQLException e) {
      log("Impossible d'obtenir la connexion à la base de données", e);
    }
  }

  // Obtenir la vraie connexion à partir du détenteur.
  con = holder.getConnection();
}

// Utiliser la connexion.
try {
  Statement stmt = con.createStatement();
  stmt.executeUpdate(
    "UPDATE INVENTORY SET STOCK = (STOCK - 10) WHERE PRODUCTID = 7");
  stmt.executeUpdate(
    "UPDATE SHIPPING SET SHIPPED = (SHIPPED + 10) WHERE PRODUCTID = 7");

  // Débiter la carte de crédit et valider la transaction dans
  // une autre servlet.
  res.sendRedirect(res.encodeRedirectURL(
    req.getContextPath() + "/servlet/CreditCardHandler"));
}
catch (Exception e) {
  // Toute erreur conduit à défaire la transaction.
  try {
    con.rollback();
    session.removeAttribute("servletapp.connection");
  }
```

Exemple 9-10. Associer une connexion à chaque session (suite)

```
      catch (Exception ignored) { }
      out.println(
        "Échec de la commande. Veuillez contacter le support technique.");
    }
  }
}
```

Au lieu de lier directement une connexion à la session, nous avons créé une simple classe de détention qui implémente l'interface `HttpSessionBindingListener`. Nous avons fait ce choix car les connexions à une base de données sont les ressources les plus limitées dans une application JDBC et nous voulons être sûrs qu'elles sont libérées proprement quand on n'en a plus besoin. La classe enveloppe nous permet aussi de défaire tout changement non validé. Si un utilisateur quitte notre magasin en ligne avant de payer, sa transaction est défaite lorsque la session expire.

Le stockage des connexions dans des sessions demande une analyse soignée des besoins de l'application. La plupart des serveurs de base de données d'entrée et de milieu de gamme peuvent gérer un maximum de 100 connexions ; les bases de données de bureau comme Microsoft Access saturent même bien avant.

Une servlet Livre d'or

Pour faciliter l'assimilation de nos concepts autour des bases de données, examinons une servlet réelle. L'*exemple 9-11* montre le code d'une servlet « Livre d'or » s'appuyant sur une base de données. Cette servlet gère une page web sur laquelle les visiteurs peuvent entrer des commentaires et voir ceux que les autres visiteurs ont écrits. Une copie d'écran est donnée à la *figure 9-4*. Tous les commentaires des visiteurs sont détenus dans une base de données. Pour l'accès à cette base, la servlet utilise plusieurs techniques décrites dans ce chapitre y compris la réserve de connexions, une instruction préparée et la lecture des informations de configuration de la base de données à partir des paramètres d'initialisation du contexte à l'aide d'un `ContextProperties`. Elle étend même la classe `com.oreilly.servlet.CacheHttpServlet` du chapitre 4 afin d'optimiser les performances de sa sortie.

Exemple 9-11. Merci de signer

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.CacheHttpServlet;

public class Guestbook extends CacheHttpServlet {

  static final String SELECT_ALL =
    "SELECT name, email, cmt, id FROM guestlist ORDER BY id DESC";
```

Exemple 9-11. Merci de signer (suite)

```
static final String INSERT =
  "INSERT INTO guestlist (id, name, email, cmt) " +
  "VALUES (?, ?, ?, ?)";

private long lastModified = 0; // Date de dernière modification de la base.
private ConnectionPool pool;

// Obtenir un pointeur sur une réserve de connexions.
public void init() throws ServletException {
  try {
    ServletContext context = getServletContext();
    synchronized (context) {
      // Une réserve peut déjà être sauvée dans un attribut du contexte.
      pool = (ConnectionPool) context.getAttribute("pool");
      if (pool == null) {
        // Construire une réserve à partir des paramètres d'initialisation
        // du contexte connection.driver, connection.url, user, password,
        // etc
        pool = new ConnectionPool(new ContextProperties(context), 3);
        context.setAttribute("pool", pool);
      }
    }
  }
  catch (Exception e) {
    throw new UnavailableException(
    "Impossible de trouver une réserve de connexions dans le contexte : " +
    e.getMessage());
  }
}

// Afficher les entrées courantes, puis demander la saisie d'une nouvelle
// entrée.
public void doGet(HttpServletRequest req, HttpServletResponse res)
                          throws ServletException, IOException {
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  printHeader(out);
  printForm(out);
  printMessages(out);
  printFooter(out);
}

// Ajouter une nouvelle entrée, puis revenir à doGet().
public void doPost(HttpServletRequest req, HttpServletResponse res)
                           throws ServletException, IOException {
  handleForm(req, res);
  doGet(req, res);
}
```

Exemple 9-11. Merci de signer (suite)

```
private void printHeader(PrintWriter out) {
  out.println("<HTML><HEAD><TITLE>Livre d'or</TITLE></HEAD>");
  out.println("<BODY>");
}

private void printForm(PrintWriter out) {
  out.println("<FORM METHOD=POST>");  // Se l'envoyer.
  out.println("<B>Veuillez envoyer votre commentaire :</B><BR>");
  out.println("Votre nom : <INPUT TYPE=TEXT NAME=name><BR>");
  out.println("Votre e-mail : <INPUT TYPE=TEXT NAME=email><BR>");
  out.println("Commentaire : <INPUT TYPE=TEXT SIZE=50 NAME=comment><BR>");
  out.println("<INPUT TYPE=SUBMIT VALUE=\"Envoyer\"><BR>");
  out.println("</FORM>");
  out.println("<HR>");
}

// Lire les messages dans la base de données et les afficher.
private void printMessages(PrintWriter out) throws ServletException {
  String name, email, comment;

  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;

  try {
    con = pool.getConnection();
    stmt = con.createStatement();
    rs = stmt.executeQuery(SELECT_ALL);

    while (rs.next()) {
      name = rs.getString(1);
      if (rs.wasNull() || name.length() == 0) name = "Utilisateur inconnu";
      email = rs.getString(2);
      if (rs.wasNull() || email.length() == 0) name = "E-mail inconnu";
      comment = rs.getString(3);
      if (rs.wasNull() || comment.length() == 0) name = "Aucun commentaire";
      out.println("<DL>");
      out.println("<DT><B>" + name + "</B> (" + email + ") a dit");
      out.println("<DD><PRE>" + comment + "</PRE>");
      out.println("</DL>");
    }
  }
  catch (SQLException e) {
    throw new ServletException(e);
  }
  finally {
    try {
      if (stmt != null) stmt.close();
    }
```

Exemple 9-11. Merci de signer (suite)

```
      catch (SQLException ignored) { }
      pool.returnConnection(con);
    }
  }

  private void printFooter(PrintWriter out) {
    out.println("</BODY>");
  }

  // Sauver le nouveau commentaire dans la base de données.
  private void handleForm(HttpServletRequest req,
                          HttpServletResponse res) throws ServletException {
    String name = req.getParameter("name");
    String email = req.getParameter("email");
    String comment = req.getParameter("comment");

    Connection con = null;
    PreparedStatement pstmt = null;
    try {
      con = pool.getConnection();
      // Utiliser une instruction préparée pour l'échappement automatique
      // des chaînes de caractères.
      pstmt = con.prepareStatement(INSERT);
      long time = System.currentTimeMillis();
      pstmt.setString(1, Long.toString(time));
      pstmt.setString(2, name);
      pstmt.setString(3, email);
      pstmt.setString(4, comment);
      pstmt.executeUpdate();
    }
    catch (SQLException e) {
      throw new ServletException(e);
    }
    finally {
      try {
        if (pstmt != null) pstmt.close();
      }
      catch (SQLException ignored) { }
      pool.returnConnection(con);
    }

    // Signaler que nous avons une nouvelle date de dernière modification.
    lastModified = System.currentTimeMillis();
  }

  public long getLastModified(HttpServletRequest req) {
    return lastModified;  // Support de CacheHttpServlet.
  }
}
```

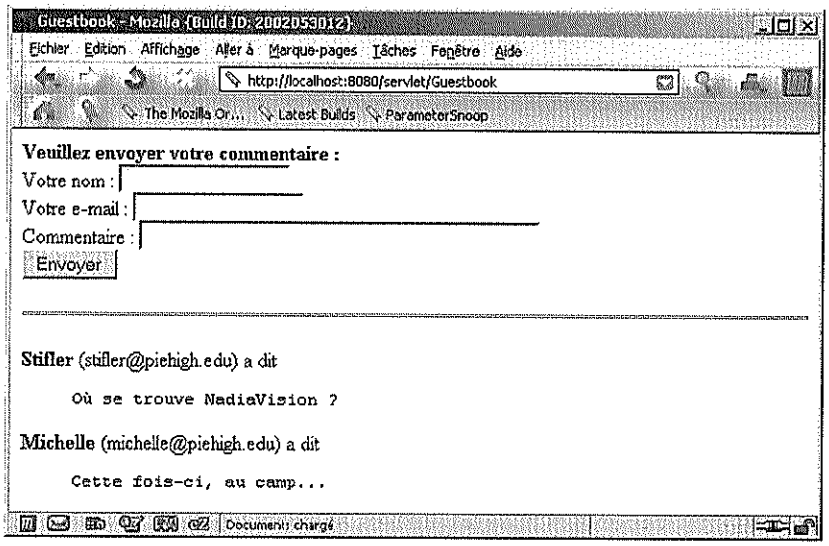


Figure 9-4. Affichage du livre d'or

Les instructions SQL d'accès à la base de données sont données au début de la classe dans des variables statiques finales. La séparation du SQL facilite les modifications futures.

La méthode `init()` obtient (ou crée) une réserve de connexions pour cette servlet. Elle commence par voir s'il n'existe pas déjà une réserve sauvegardée dans un attribut de `ServletContext` sous le nom `pool`. Si cet attribut n'existe pas, la méthode `init()` construit une nouvelle réserve, en utilisant la classe `ContextProperties` pour spécifier les information de configuration de la base de données, et la sauvegarde dans le contexte. Pour être franc, une réserve de connexions n'est pas vraiment nécessaire dans cette servlet car il n'y a pas de transactions et le trafic *via* un seul `Connection` réutilisé ne serait pas trop important. Dans des cas comme celui-ci, l'emploi d'une réserve de connexions n'est pas insensé à condition qu'elle permette de retourner un objet `Connection` partagé pour une utilisation simultanée par plusieurs servlets.

La méthode `doGet()` affiche l'en-tête, un formulaire demandant le commentaire de l'utilisateur, les commentaires précédents pris dans la base de données et une fin. Des techniques de création HTML plus élégantes pour des cas comme celui-ci sont montrées à partir du chapitre 14, *L'environnement Tea*. Pour le moment, cela nous suffit.

La méthode `printMessages()` utilise la réserve pour obtenir un `Connection`, puis elle exécute la requête `SELECT_ALL` à l'aide d'un `Statement` obtenu à l'aide du `Connection`. Pour chaque ligne du `ResultSet`, elle affiche une entrée `<DL>`. Dans le bloc `finally`, le `Connection` est replacé dans la réserve.

La méthode `doPost()` est appelée quand un utilisateur soumet ses commentaires *via* le formulaire généré par `doGet()`. Cette méthode appelle `handleForm()` pour stocker le commentaire dans la base de données, puis elle passe à `doGet()` pour afficher la page. La méthode `handleForm()` récupère les paramètres `name`, `email` et `comment`, et elle utilise un `PreparedStatement` et l'instruction `INSERT` pour sauvegarder ces informations dans la base de données. En utilisant un `PreparedStatement` pour sauvegarder les chaînes, on

applique automatiquement l'échappement sur les caractères spéciaux. Une estampille horaire est aussi sauvegardée afin de pouvoir trier les commentaires et des affichages futurs.

Pour finir, la méthode `handleForm()` donne à `lastModified` la valeur de la date courante. Cela permet à la méthode `getLastModified()` de retourner la date à laquelle la base de données a été modifiée pour la dernière fois. Puisque cette servlet étend `CacheHttpServlet`, l'information de dernière modification sera utilisée par la classe mère pour gérer un cache de sa sortie, ne modifiant le cache qu'après une mise à jour de la base de données.[1]

Techniques JDBC avancées

Maintenant que nous avons vu les bases, examinons quelques techniques avancées qui utilisent les servlets et JDBC. Tout d'abord, nous allons voir comment les servlets peuvent accéder aux procédures stockées d'une base de données. Ensuite, nous regarderons comment les servlets peuvent récupérer dans une base de données des types de données complexes, comme les données binaires (images, applications, etc.), de grandes quantités de texte, ou même du code de manipulation de base de données exécutable.

Procédures stockées

La plupart des SGBDR incluent une forme de langage de programmation interne, par exemple le PL/SQL d'Oracle. Ces langages permettent aux développeurs de bases de données d'embarquer du code applicatif procédural directement dans une base de données et ensuite de l'appeler à partir d'une autre application. Les langages de programmation des SGBDR sont souvent bien adaptés à l'exécution de certaines actions de base de données ; plusieurs installations de bases de données existantes disposent de procédures stockées utiles déjà écrites et prêtes à être utilisées. La plupart des présentations de JDBC ont tendance à ne pas parler de ce sujet, nous le présentons brièvement ici.

Le code suivant est une procédure stockée écrite en PL/SQL d'Oracle :

```
CREATE OR REPLACE PROCEDURE sp_interest
(id IN INTEGER
bal IN OUT FLOAT) IS
BEGIN
SELECT balance
INTO bal
FROM accounts
WHERE account_id = id;

bal := bal + bal * 0.03;
```

1. Rappelez-vous cependant que les requêtes POST n'utilisent pas le cache car elles ne sont pas idempotentes. Cela a un effet secondaire intéressant pour l'exemple `Guestbook`. Si vous recharger la page après avoir soumis un nouveau commentaire, vous verrez que la génération de page prend un certain temps même s'il s'agit de la page que vous avez déjà vue. En effet, l'envoi était une requête POST dont la sortie n'a pas été mise dans le cache.

```
UPDATE accounts
SET balance = bal
WHERE account_id = id;

END;
```

Cette procédure exécute une instruction SQL, effectue un calcul et exécute une autre instruction SQL. Il serait simple d'écrire le SQL qui fait cela (en fait, l'exemple de transaction précédent fait quelque chose de similaire), alors pourquoi s'embêter avec cela ? Voici plusieurs raisons :

- Les procédures stockées sont précompilées dans les SGBDR, elles s'exécutent donc plus rapidement que du SQL dynamique.
- Les procédures stockées s'exécutent entièrement dans le SGBDR, elles peuvent donc effectuer plusieurs requêtes et mises à jour sans trafic réseau.
- Les procédures stockées vous permettent d'écrire du code de manipulation de la base de données une fois et de l'utiliser pour plusieurs applications dans de multiples langages.
- Les changements dans la structure des tables ne demandent des changements que dans les procédures stockées qui y accèdent ; les applications utilisant la base de données ne sont pas touchées.
- La plupart des bases de données les plus anciennes ont déjà énormément de code écrit sous forme de procédures stockées ; ce serait bien de pouvoir exploiter ce travail.

La procédure PL/SQL de notre exemple prend une valeur d'entrée, dans ce cas un identifiant de compte, et retourne une balance mise à jour. Alors que chaque base de données a sa propre syntaxe d'accès aux procédures stockées, JDBC crée une séquence d'échappement standardisée pour accéder aux procédures stockées en utilisant la classe `java.sql.CallableStatement`. La syntaxe pour une procédure qui ne retourne pas un résultat est `{call nom-de-procédure(?,?)}`. La syntaxe pour une procédure qui retourne une valeur résultat est `{? = call nom-de-procédure(?,?)}`. Les paramètres entre parenthèses sont optionnels.

L'utilisation de la classe `CallableStatement` est similaire à celle de la classe `PreparedStatement` :

```
CallableStatment cstmt = con.prepareCall("{call sp_interest(?,?)}");
cstmt.registerOutParameter(2, java.sql.Types.FLOAT);
cstmt.setInt(1, accountID);
cstmt.execute();
out.println("Nouvelle balance : " + cstmt.getFloat(2));
```

Ce code commence par créer un `CallableStatement` en utilisant la méthode `prepareCall()` de `Connection`. Cette procédure stockée ayant un paramètre de sortie, elle utilise la méthode `registerOutParameter()` de `CallableStatement` pour indiquer que ce paramètre est un paramètre de sortie de type `FLOAT`. Enfin, le code exécute la procédure stockée et utilise la méthode `getFloat()` de `CallableStatement` pour afficher la nouvelle balance. Les méthodes `getXXX()` de l'interface `CallableStatement` sont similaires à celles de l'interface `ResultSet`.

Données binaires et livres

La plupart des bases de données supportent des types de données pour gérer les chaînes de texte ayant une taille jusqu'à plusieurs gigaoctets, ainsi que des informations binaires comme les fichiers multimédia. Chaque base de données gère différemment ces types de données, mais les méthodes JDBC pour les retrouver sont standard. La méthode `getAsciiStream()` de `ResultSet` prend en charge les textes longs ; `getBinaryStream()` fonctionne pour les grands objets binaires. Chacune de ces méthodes retourne un `InputStream`.

Le support des grands types de données est l'une des sources les plus classiques des problèmes avec JDBC. Assurez-vous de bien tester vos pilotes, en utilisant les plus grands morceaux de données que votre application rencontrera. Le pilote JDBC d'Oracle est particulièrement sujet aux erreurs dans ce domaine.

Voici le code d'une servlet de panneau d'affichage qui montre la lecture d'une longue chaîne ASCII. Nous pouvons supposer que les connexions, les instructions, et ainsi de suite, ont déjà été créées :

```
try {
  ResultSet rs = stmt.executeQuery(
    "SELECT TITLE, SENDER, MESSAGE FROM MESSAGES WHERE MESSAGE_ID = 9");
  if (rs.next()) {
    out.println("<H1>" + rs.getString("title") + "</H1>");
    out.println("<B>De :</B> " + rs.getString("sender") + "<BR>");
    BufferedReader msgText = new BufferedReader(
      new InputStreamReader(rs.getAsciiStream("message")));
    while (msgText.ready()) {
      out.println(msgText.readLine());
    }
  }
}
catch (SQLException e) {
  // La signaler.
}
```

Pendant qu'elle lit à partir du `InputStream`, cette servlet ne récupère pas la valeur des autres colonnes du résultat. C'est important car appeler n'importe quelle autre méthode `getXXX()` de `ResultSet` ferme le `InputStream`.

Les données binaires peuvent être retrouvées de la même manière en utilisant `ResultSet.getBinaryStream()`. Dans ce cas, nous devons spécifier le bon type de contenu et écrire la sortie sous forme d'octets. L'*exemple 9-12* montre une servlet qui retourne un fichier GIF chargé à partir d'une base de données.

Exemple 9-12. Lire une image GIF binaire à partir d'une base de données

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DBGifReader extends HttpServlet {
```

Exemple 9-12. Lire une image GIF binaire à partir d'une base de données (suite)

```
  Connection con;

  public void init() throws ServletException {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      con = DriverManager.getConnection("jdbc:odbc:imagedb", "user", "passwd");
    } catch (ClassNotFoundException e) {
      throw new UnavailableException("Impossible de charger JdbcOdbcDriver");
    }
    catch (SQLException e) {
      throw new UnavailableException(
        "Impossible d'obtenir la connexion à la base de données");
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    try {
      res.setContentType("image/gif");
      ServletOutputStream out = res.getOutputStream();

      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
        "SELECT IMAGE FROM PICTURES WHERE PID = " + req.getParameter("PID"));

      if (rs.next()) {
        BufferedInputStream gifData =
          new BufferedInputStream(rs.getBinaryStream("image"));
        byte[] buf = new byte[4 * 1024];  // Tampon de 4K.
        int len;
        while ((len = gifData.read(buf, 0, buf.length)) != -1) {
          out.write(buf, 0, len);
        }
      } else {
        res.sendError(res.SC_NOT_FOUND);
      }
    } catch(SQLException e) {
      // La signaler.
    }
  }
}
```

API JDBC avancée

En plus des méthodes du noyau de JDBC et des techniques décrites dans ce chapitre, il existe d'autres API pour les bases de données facilitant l'écriture d'applications complexes accédant à des bases de données. JDBC 2.0 est une telle API. Il s'agit de la version après JDBC 1.2 présentée dans ce chapitre.

Au moment de l'écriture de ce livre, JDBC 2.0 n'est pas encore largement implémenté et utilisé. Pour employer JDBC 2.0, il faut que la base de données supporte ses fonctionnalités avancées et qu'un pilote y donne accès. Vous trouverez le support de JDBC 2.0 dans tous les serveurs J2EE car c'est une obligation pour tout serveur qui se veut compatible avec J2EE. Par exemple, l'implémentation de référence de J2EE de Sun vient avec une copie d'évaluation de la base de données Java Cloudscape.

JDBC 2.0 est constitué de deux parties. Une première partie réside dans le paquetage `java.sql` de Java 2 et elle est doit être supportée par toute base de données et pilote compatibles avec JDBC 2.0. L'autre partie réside dans le paquetage `javax.sql`. Ce paquetage optionnel (*Optional Package*) offre plusieurs fonctionnalités qui peuvent être implémentées par une base de données et un pilote compatibles avec JDBC 2.0, mais ce n'est pas une obligation.

Parmi les améliorations dans le paquetage `java.sql`, on trouve les ensembles de résultat défilants et modifiables. Ces deux fonctionnalités permettent de déplacer le curseur du `ResultSet` de façon arbitraire, vers l'avant ou vers l'arrière (au lieu de juste une ligne en avant à la fois), et de modifier, insérer et supprimer des lignes à l'emplacement du curseur. On a aussi ajouté à ce paquetage le support des nouveaux types de données de SQL3 (dont `BLOB` pour les grands objets binaires, `CLOB` pour les grands objets textuels et `ARRAY` pour les tableaux) et les mises à jour par lot permettant d'envoyer au serveur plusieurs instructions de mise à jour en une seule requête réseau.

Le paquetage optionnel `javax.sql` inclut plusieurs fonctionnalités particulièrement importantes pour les applications d'entreprise. Une des ces fonctionnalités est le support de la recherche JNDI. Cela permet de retrouver une connexion à une base de données par son nom dans un serveur JNDI. Une autre fonctionnalité est la réserve de connexions intégrée, quelque chose qui a sévèrement modifié l'API mais qui permet de retrouver une connexion déjà existante. Une fonctionnalité optionnelle extrêmement importante de JDBC 2.0 est le support des transactions distribuées. Elle permet de créer des transactions qui se répartissent sur plusieurs bases de données séparées, les mises à jour de l'ensemble des bases de données se faisant dans une opération atomique. Enfin, JDBC 2.0 inclut le support des objets `RowSet` qui enveloppent les données des lignes d'un `ResultSet`. Ces objets permettent de mettre facilement des données dans un cache et facilitent leur transfert sur un réseau. Ils offrent aussi un accès standardisé à toute source de données de type tableau comme une feuille de tableau ou un fichier à plat.

Pour plus d'informations sur les fonctionnalités de JDBC 2.0, consultez les livres *JDBC et Java, Guide du programmeur*, de George Reese (Éditions O'Reilly) et *JDBC API Tutorial and Reference* de Seth White et *al.* (Addison-Wesley), ainsi que les tutoriels en ligne disponibles à *http://java.sun.com/products/jdbc*.

Enfin, pour ceux qui préfèrent se placer à un niveau plus élevé que celui fournit par JDBC, la société Clear Ink sponsorise une bibliothèque en *open source* (licence de type Apache) appelée Village. Elle se place au-dessus de JDBC 1.2 et elle présente une API qui simplifie les interactions avec la base de données. Elle est directement modélisée d'après le produit commercial dbKona de BEA/WebLogic. D'autres personnes ont développé au-dessus de Village afin de produire une version plus robuste appelée Town. Village et Town sont disponibles à *http://www.working-dogs.com*. (Rien encore à propos de City.)

10

Communication applet-servlet

Ce chapitre explique plusieurs techniques permettant aux applets de communiquer avec des servlets. Nous allons aborder ce sujet par un angle un peu différent de celui auquel vous pourriez penser. Au lieu de supposer l'existence d'une applet et d'une servlet qui doivent communiquer, nous allons supposer qu'il existe une applet qui doit parler à une entité sur le serveur et explorer pourquoi cette entité doit être une servlet.

Pour démarrer, imaginons des applets ayant besoin de communiquer avec le serveur. Il en existe plusieurs bons exemples. Voyez l'applet d'administration qui gère Java Web Server. Pensez à son fonctionnement : elle s'exécute sur le client, mais elle configure le serveur. Pour cela, l'applet et le serveur doivent être en communication constante. Comme autre exemple, voyez une applet de discussion. Un client dit quelque chose, et tous les autres le voit. Comment cela fonctionne-t-il ? Il n'y a sûrement pas une communication d'applet à applet. En fait, chaque applet dépose son message sur un serveur central qui prend en charge la mise à jour des autres clients. Enfin, imaginez une applet qui suit le cours d'un ensemble d'actions et offre une mise à jour en continu. Comment cette applet connaît-elle la valeur courante des actions et, plus important, comment est-elle informée de leur changement ? La réponse est qu'elle discute avec son serveur.

Options de communication

Notre intérêt pour la bourse grimpant avec le CAC, continuons donc avec cet exemple d'applet hypothétique de suivi d'actions. Nous devons vous avertir maintenant que cet exemple va rester théorique. Nous ne l'utiliserons que comme un point de référence pour discuter des problèmes soulevés par la communication applet-serveur. Mais ne vous inquiétez pas, il y a une foule d'autres exemples de code plus loin dans ce chapitre qui démontrent les techniques discutées ici, des exemples un peu plus simples.

Cette applet de suivi d'actions a besoin de s'alimenter à partir d'une certaine machine serveur. Si l'on suppose que c'est une applet normale non sûre, il n'y a qu'une possibilité : cette machine est celle à partir de laquelle elle a été téléchargée. Tout essai de connexion à une autre machine résultera en une `SecurityException`. Supposons

donc que l'applet prend ses informations sur la machine à partir de laquelle elle a été téléchargée[1]. Il reste encore la question : comment l'applet et le serveur communiquent-il ?

Applets sûres et non-sûres

Lorsqu'une applet Java est embarquée dans une page web, un navigateur peut la télécharger et l'exécuter automatiquement. Si vous y réfléchissez, c'est quelque chose de très dangereux. Donc, pour protéger le client, le JDK 1.0 supposait que les applets n'étaient pas sûres et les exécutait sous le contrôle d'un `SecurityManager` qui limitait beaucoup leurs possibilités. Par exemple, le gestionnaire de sécurité garantit que les applets ne peuvent écrire sur le système de fichiers de l'utilisateur, lire certaines propriétés du système, accepter des connexions par sockets entrantes, ou établir des connexions par sockets sortantes à toute machine autre que le serveur d'origine. Cela protège le client, mais limite l'utilité des applets.

Par conséquent, le JDK 1.1 a introduit le concept d'applets sûres, des applets qui peuvent fonctionner comme des applications normales avec un plein accès à la machine du client. Pour qu'une applet soit sûre, elle doit avoir été signée numériquement par une personne ou une société auxquelles le client fait confiance (comme marqué dans le navigateur du client). La signature authentifie l'origine de l'applet et garantit l'intégrité pendant le transfert, le client sait donc que le code de l'applet n'a pas été modifié. Cela autorise des applets plus productives, mais c'est une approche tout ou rien.

Pour donner plus de contrôle au client, le JDK 1.2 a introduit un système de contrôle d'accès à granularité fine. Dans ce système, on peut faire partiellement confiance à une applet signée numériquement, lui donnant certaines possibilités mais pas un pouvoir absolu sur le système. Cela permet d'accorder des privilèges faibles (comme écrire dans un seul répertoire) à des applets provenant de sources inconnues, sans leur permettre de nettoyer le disque dur du client. Les fournisseurs de navigateurs sont lents à fournir un support intégré des nouvelles versions du JDK, mais il est heureusement possible de mettre à jour la JVM d'un navigateur en installant le Java Plug-In, un produit disponible gratuitement à *http://java.sun.com/products/plugin*.

HTTP et connexions par socket

Avant le JDK 1.1 et les servlets, il y avait deux options pour la communication applet-serveur :

- Que l'applet établisse une connexion HTTP avec un programme CGI sur la machine serveur. L'applet agit comme un navigateur et demande une page, analy-

1. Vous pourriez vous demander comment la machine serveur obtient elle-même les informations sur les actions. Pour cet exemple, c'est par magie.

sant la réponse pour son propre usage. L'applet peut fournir des informations en utilisant une chaîne d'interrogation ou des données POST, et peut recevoir des informations de la page retournée.

- Que l'applet établisse une connexion par socket à un serveur non-HTTP tournant sur la machine serveur. Le serveur non-HTTP peut écouter sur un port particulier et communiquer avec l'applet en utilisant n'importe quel protocole personnalisé.

Chacune de ces approches a ses avantages et ses inconvénients. Utiliser une connexion HTTP entre une applet et un programme CGI fonctionne bien car :

- C'est facile à écrire. L'applet peut exploiter les classes `java.net.URL` et `java.net.URLConnection` pour gérer le canal de communication, et le programme CGI peut être écrit comme tous les autres.
- Cela fonctionne même pour des applets s'exécutant derrière un firewall. La plupart des firewalls autorisent les connexions HTTP mais interdisent les connexions par socket.
- Cela permet à une applet Java de communiquer avec un programme écrit avec n'importe quel langage. Le programme CGI n'a pas besoin d'être écrit en Java. Il peut l'être en Perl, C, C++ ou tout autre langage.
- Cela fonctionne avec des applets écrites avec le JDK 1.0 et donc avec tous les navigateurs supportant Java.
- Cela autorise une communication sûre. Une applet peut communiquer avec un serveur sûr en utilisant le protocole chiffré HTTPS (HTTP + SSL).
- Le programme CGI peut être utilisé par les navigateurs et les applets. Dans le cas de notre exemple de suivi d'actions, le programme CGI peut avoir deux rôles, agissant aussi comme base arrière pour un service de cotation d'actions basé sur des formulaires HTML. Il est ainsi très facile pour une servlet d'exploiter les programmes CGI existants.

Mais la connexion HTTP à un programme CGI peut aussi avoir certaines problèmes :

- C'est lent. En raison du principe de requête/réponse de HTTP, l'applet et le programme CGI ne peuvent pas communiquer de façon interactive. Ils doivent rétablir un nouveau canal de communication pour chaque requête et réponse. De plus, il y a le délai standard de lancement et d'initialisation du programme CGI pour traiter une requête.
- Cela nécessite habituellement que les requêtes soient construites comme un tableau de paires nom/valeur. Par exemple, lorsque notre applet de suivi d'actions demande le plus haut cours du jour de l'action Sun Microsystems, elle doit poser la question avec une chaîne d'interrogation comme `stock=sunw& query=dailyhi`.
- Cela impose un formatage de toutes les réponses avec un standard arbitraire décidé par les deux parties. Par exemple, lorsque notre applet de suivi d'actions reçoit la réponse qui contient le cours le plus haut du jour de l'action, elle doit savoir exactement comme analyser les données. Le prix retourné commence-t-il par un signe dollar ? La réponse inclut-elle l'heure à laquelle le cours le plus haut est survenu ? Et si oui, cette heure est-elle spécifiée et dans quel format ?

- Seule l'applet peut initier la communication. Le programme CGI doit attendre passivement la requête de l'applet avant de pouvoir répondre. Si le prix de l'action change, l'applet ne le saura que lorsqu'elle posera la bonne question.

Une applet et un serveur peuvent aussi communiquer en créant une connexion par socket entre l'applet et un processus serveur non-HTTP. Par rapport à l'approche basée sur HTTP, en voici les avantages :

- La communication peut être bidirectionnelle et permanente. L'applet et le serveur peuvent utiliser la même socket (ou même plusieurs sockets) pour communiquer de façon interactive. Pour des raisons de sécurité, l'applet doit toujours initier la connexion en se connectant à une socket serveur sur le machine serveur, mais après l'établissement de la connexion de la socket, les deux parties peuvent écrire sur la socket à tout moment. Notre applet de suivi d'actions peut ainsi recevoir la mise à jour du prix des actions dès qu'il est disponible.
- Un programme plus efficace peut être exécuté sur le serveur. Il est possible d'écrire le serveur non-HTTP afin qu'il traite une requête immédiatement sans lancer un programme CGI externe pour faire le travail.

Mais une connexion par socket a aussi les inconvénients suivants par rapport à l'approche basée sur HTTP :

- Elle ne fonctionne pas pour les applets derrière un firewall. La plupart des firewalls n'autorisent pas les connexions par socket et ils interdisent donc cette forme de communication applet-serveur. C'est pourquoi ce mécanisme ne doit être utilisé que lorsque l'on est sûr que l'applet ne sera pas derrière un firewall, par exemple dans une application intranet.
- Le code qui tourne sur le serveur peut être assez complexe à écrire. Il doit toujours y avoir un processus (comme un serveur de cotation d'actions) en train d'écouter sur un port bien connu de la machine serveur. Le développement d'une telle application en Java est plus facile qu'en C++, mais sans être triviale.
- On peut avoir besoin de développer un protocole personnalisé. L'applet et le serveur doivent définir le protocole qu'ils utilisent pour communiquer. Bien que ce protocole puisse être plus simple et plus efficace que HTTP, il doit souvent être développé spécialement.
- Un navigateur web ne peut pas se connecter facilement à un serveur non-HTTP. Les navigateurs parlent HTTP ; ils ne peuvent communiquer avec un serveur non-HTTP.

L'approche historique standard fut que les applets utilisent HTTP pour se connecter à un programme CGI sur le serveur. C'est facile et cela fonctionne pour tous les types de navigateurs, même ceux qui se trouvent derrière un firewall. L'utilisation des connexions par socket a été en général réservée aux situations dans lesquelles c'est absolument nécessaire, comme lorsque l'applet et le serveur ont besoin d'une communication bidirectionnelle. Et, même dans ces cas, il est souvent possible d'utiliser des connexions HTTP pour simuler une communication bidirectionnelle afin de traverser les firewalls, comme nous le verrons plus tard dans un exemple.

Servlets et sérialisation d'objets

L'introduction récente des servlets Java et de la sérialisation d'objets a donné une nouvelle vie à ces techniques de communication applet-serveur traditionnelles. Les servlets commencent lentement à remplacer les programmes CGI, améliorant les performances de la communication applet-serveur basée sur HTTP et lui permettant d'être très fréquente. Bien qu'il soit vrai que dans un cas général l'applet et la servlet doivent prendre le temps de rétablir leur connexion à chaque requête et réponse, l'applet n'a plus besoin d'attendre que le serveur lance un programme CGI qui traite chacune des requêtes répétées.

La sérialisation d'objets Java a simplifié les problèmes liés au formatage des réponses. Les applets et les servlets étant écrites en Java, il est naturel qu'elles communiquent en échangeant des objets Java. Par exemple, lorsque notre applet hypothétique de suivi d'actions demande à notre servlet fournisseur le plus haut cours du jour de l'action Sun, elle peut recevoir la réponse sous forme d'un objet `StockPrice` sérialisé. À partir de là, elle peut obtenir la valeur du plus haut cours sous forme d'un `float` et l'heure correspondante comme un `Date`. C'est pratique et assure un contrôle de type facile. Mais attention, la sérialisation d'objet ne fonctionne qu'avec les applets s'exécutant dans des navigateurs qui supportent la version 1.1 et supérieure du JDK.

JDBC, RMI et un peu de CORBA

Le JDK 1.1 inclut deux fonctionnalités supplémentaires qui ont un impact sur la communication applet-serveur : JDBC et RMI. L'API JDBC (*Java DataBase Connectivity*), présentée au chapitre 9, *Se connecter aux bases de données*, permet à un programme Java de se connecter à une base de données relationnelle sur la même machine ou sur une autre machine. Les applets Java écrites avec le JDK 1.1 peuvent utiliser JDBC pour communiquer avec une base de données sur le serveur. Cette communication spécifique ne nécessite pas en général une communication applet-servlet. Cependant, pour une applet (spécialement une écrite avec le JDK 1.0), il est souvent plus facile de renoncer à une connexion directe avec la base de données (ou à un proxy sur le serveur web) et de plutôt se connecter à une servlet qui gère la communication avec la base de données pour l'applet (comme expliqué dans l'encart *Servlets dans le tiers intermédiaire* au chapitre 9). Par exemple, une applet qui veut rechercher l'adresse d'une personne peut se connecter à une servlet en utilisant HTTP, passer le nom de la personne en utilisant des paramètres HTTP et ensuite recevoir l'adresse, soit dans une chaîne au format spécial, soit par un objet sérialisé. Cette utilisation de la communication applet-servlet a tendance à se reposer sur des protocoles existants comme HTTP, nous n'allons donc pas la détailler ici.

L'API RMI (*Remote Method Invocation*) permet à une applet d'invoquer des méthodes d'un autre objet Java s'exécutant sur la machine serveur et, dans certains cas, elle permet aussi à l'objet sur la machine serveur d'invoquer les méthodes de l'applet. Les avantages de RMI pour la communication applet-serveur sont attrayants :

- Les applets et les objets du serveur peuvent communiquer en utilisant un paradigme orienté objet élégant de haut niveau. Les requêtes peuvent se faire sous forme d'invocations de méthodes, si nécessaire en passant en paramètre des objets sérialisés. Les réponses peuvent être reçues sous forme d'objets sérialisés ou même de référence à d'autres objets distants. Mais le fait même d'utiliser les mots *requêtes*

et *réponses* montre que l'on a trop utilisé HTTP ! Avec RMI, il n'y a ni requêtes ni réponses, juste des invocations de méthodes. Pour revenir à notre exemple de suivi d'actions, l'applet peut obtenir le plus haut cours du jour pour l'action Sun en appelant `sunw.getDailyHigh()`, où `sunw` est un objet Java qui existe sur le serveur.

- Les objets du serveur peuvent appeler les méthodes de l'applet. Par exemple, dans l'applet de suivi d'actions, le serveur peut notifier aux applets intéressées que le prix d'une action a changé en appelant `applet.update(stock)`.
- Les firewalls peuvent ne pas être un problème (bien que RMI ne les aime pas trop, et les navigateurs courants ne le supportent pas très bien). La couche de transport RMI repose normalement sur des connexions par sockets directes pour faire son travail. Cependant, lorsqu'une applet se trouve derrière un firewall, ses connexions par sockets échouent. Dans ce cas, la couche de transport RMI peut automatiquement commencer à opérer avec le protocole HTTP[2].

Mais il y a un coût. Le surcoût de HTTP affecte les performances et son paradigme de requête/réponse n'accepte pas les rappels. Les inconvénients de RMI sont tout aussi importants :

- Il est complexe. La communication RMI utilise des classes stub et squelette spécifiques pour chaque objet distant, et impose un registre de noms à partir duquel chaque client peut obtenir des références aux objets distants.
- Il n'est supporté que par quelques navigateurs. De tous les navigateurs populaires disponibles, au moment de l'écriture de ce livre, seul Netscape Navigator 4 et ultérieurs supportent RMI. Les versions précédentes du navigateur de Netscape et toutes les versions d'Internet Explorer de Microsoft ne supportent RMI que par l'installation d'une extension. (Cependant, les classes RMI étant du Java pur, certaines personnes ont réussi à ajouter RMI à Internet Explorer en tant que bibliothèque tierce.)
- Seuls les clients Java peuvent l'utiliser. L'objet du serveur ne peut pas être partagé par un navigateur web ou même un client C++.

Pour plus d'informations sur la programmation RMI, lisez *Programmation réseau avec Java* d'Elliotte Rusty Harold (Éditions O'Reilly) et *Java Distributed Computing* de Jim Farley (O'Reilly & Associates).

CORBA (*Common Object Request Broker Architecture*) est une technologie similaire à RMI qui autorise la communication entre des objets distribués écrits dans différents langages. Avec CORBA et son protocole de communication IIOP (*Internet Inter-ORB Protocol*), un client C++ peut communiquer avec une servlet Java. L'exposé de cette solution dépasse le cadre de ce livre. Pour plus d'informations allez voir à *http://www.omg.org* et *http://java.sun.com/products/jdk/idl*.

---

2. Pour une description des propriétés système nécessaires pour qu'une application RMI cliente puisse traverser un firewall, lisez le JavaWorld Java Tip 42 de John D. Mitchell à *http://www.javaworld.com/javaworld/javatips/jw-javatip42.html*. (Non mentionnées dans cet article, mais aussi importantes, les propriétés `socksProxySet`, `socksProxyHost` et `socksProxyPort` sont nécessaires pour les proxies basés sur SOCKS.) Toutes ces propriétés système devraient être définies automatiquement par les navigateurs web, mais, malheureusement, peu d'entre eux le font, laissant leurs applets sans moyen de déterminer la bonne configuration et ne sachant comment utiliser RMI au travers d'un firewall.

Approche hybride

Maintenant que nous avons vu toutes les options, la question est : comment notre applet de suivi d'actions doit-elle communiquer avec son serveur ? La réponse est : cela dépend.

Si nous pouvons garantir que tous nos clients potentiels le supportent, l'élégance et la puissance de RMI en font le choix idéal. Mais aujourd'hui cela revient à supposer que tous vos amis apprécient vos blagues sur Star Trek. C'est possible si vous choisissez bien vos amis (ou vos clients), mais ce n'est généralement pas le cas dans le monde réel.

Quand RMI n'est pas disponible, les possibilités bidirectionnelles de la connexion par socket non-HTTP la rendent attractive. Malheureusement, cette communication bidirectionnelle devient une communication inexistante lorsque l'applet termine derrière un firewall.

Il y a toujours le vieux tracteur, la communication HTTP, facile à mettre en œuvre et fonctionnant sur tous les clients supportant Java. Et si vous pouvez garantir que le client supporte le JDK 1.1 (plus facile à garantir que le client supporte RMI), vous pouvez utiliser la sérialisation d'objet.

La meilleure solution consiste peut-être à utiliser chaque solution. Java permet de combiner les techniques de communication applet-serveur HTTP, non-HTTP et RMI, le tout dans une même application. Pourquoi vouloir faire cela ? Et bien, c'est une technique pratique lorsqu'une applet veut communiquer en utilisant RMI ou un protocole non-HTTP mais a besoin de se rabattre sur HTTP si nécessaire (comme lorsqu'elle se retrouve derrière un firewall). En utilisant le même serveur pour gérer chaque client, la logique du serveur et son état peuvent se trouver au même endroit. Bien sûr, lorsque vous contrôlez votre environnement, vous pouvez enlevez un ou plusieurs de ces protocoles. Mais n'est-il pas agréable de savoir que ce n'est pas la peine ?

Dans le cas d'une application complexe s'exécutant dans un serveur d'applications, la conception standard est de rendre disponible un objet RMI distant pour supporter les clients RMI, un auditeur de socket pour supporter les clients par socket et une servlet pour supporter les clients HTTP. Ces objets partagent tous un ensemble de classes implémentant la logique métier pour gérer les requêtes du client (un peu comme un restaurant offrant un service de vente à emporter qui prend les commandes par téléphone, fax ou e-mail). Pour la suite de ce chapitre, nous simplifierons les choses et montrerons la gestion des communications RMI, par socket et HTTP par une seule servlet. Une servlet, plusieurs protocoles d'accès.

Serveur horaire

Pour une démonstration simple de chaque technique de communication, nous allons écrire une applet qui demande à son serveur l'heure courante. L'applet commence par utiliser une connexion HTTP, ensuite une connexion par socket non-HTTP et enfin une connexion RMI. Bien sûr, une applet peut connaître l'heure à partir du système sur lequel elle s'exécute. Pour donner à cet exemple un air pratique, supposons que l'applet ait besoin d'une estampille temporelle approximative pour un certain événement et ne peut pas supposer que l'heure sur la machine cliente est correcte.

L'applet

Nous allons utiliser le même exemple tout au long de cette section. Le squelette de code pour cette applet, DaytimeApplet, est donné à l'*exemple 10-1*. Pour le moment, l'applet ne fait que créer une interface utilisateur où le temps est affiché, comme le montre la *figure 10-1*. Au fur et à mesure que nous avancerons dans cet exemple, nous implémenterons ses méthodes getDateUsingHttpText(), getDateUsingHttpObject(), getDateUsingSocketText(), getDateUsingSocketObject() et getDateUsingRMIObject().

Notez que les exemples de ce chapitre utilisent plusieurs méthodes du JDK 1.0 qui sont obsolètes dans le JDK 1.1, afin d'augmenter la portabilité. Vous verrez des avertissements de désapprobation avec les nouveaux JDK, mais vous pouvez les ignorer.

Exemple 10-1. DaytimeApplet, sans le code intéressant

```
import java.applet.*;
import java.awt.*;
import java.io.*;
import java.util.*;

public class DaytimeApplet extends Applet {

  TextField httpText, httpObject, socketText, socketObject, RMIObject;
  Button refresh;

  public void init() {
    // Construire l'interface graphique.

    setLayout(new BorderLayout());

    // À gauche, créer des étiquettes pour les divers mécanismes de
    // communication.
    Panel west = new Panel();
    west.setLayout(new GridLayout(5, 1));
    west.add(new Label("Texte HTTP : ", Label.RIGHT));
    west.add(new Label("Objet HTTP : ", Label.RIGHT));
    west.add(new Label("Texte par socket : ", Label.RIGHT));
    west.add(new Label("Objet par socket : ", Label.RIGHT));
    west.add(new Label("Objet RMI : ", Label.RIGHT));
    add("West", west);

    // À droite, créer des champs de texte pour afficher les valeurs
    // retrouvées.
    Panel center = new Panel();
    center.setLayout(new GridLayout(5, 1));

    httpText = new TextField();
    httpText.setEditable(false);
    center.add(httpText);

    httpObject = new TextField();
```

Exemple 10-1. DaytimeApplet, sans le code intéressant (suite)

```
    httpObject.setEditable(false);
    center.add(httpObject);

    socketText = new TextField();
    socketText.setEditable(false);
    center.add(socketText);

    socketObject = new TextField();
    socketObject.setEditable(false);
    center.add(socketObject);

    RMIObject = new TextField();
    RMIObject.setEditable(false);
    center.add(RMIObject);

    add("Center", center);

    // En bas, créer un bouton de mise à jour des dates.
    Panel south = new Panel();
    refresh = new Button("Mettre à jour");
    south.add(refresh);
    add("South", south);
  }

  public void start() {
    refresh();
  }

  private void refresh() {
    // Retrouver et afficher les dates.
    httpText.setText(getDateUsingHttpText());
    httpObject.setText(getDateUsingHttpObject());
    socketText.setText(getDateUsingSocketText());
    socketObject.setText(getDateUsingSocketObject());
    RMIObject.setText(getDateUsingRMIObject());
  }

  private String getDateUsingHttpText() {
    // Retrouver la date courante via une connexion HTTP à base de texte.
    return "non disponible";
  }

  private String getDateUsingHttpObject() {
    // Retrouver la date courante via une connexion HTTP à base d'objet.
    return "non disponible";
  }

  private String getDateUsingSocketText() {
    // Retrouver la date courante via une connexion par socket non-HTTP à
```

Exemple 10-1. DaytimeApplet, sans le code intéressant (suite)

```
    // base de texte.
    return "non disponible";
  }

  private String getDateUsingSocketObject() {
    // Retrouver la date courante via une connexion par socket non-HTTP à
    // base d'objet.
    return "non disponible";
  }

  private String getDateUsingRMIObject() {
    // Retrouver la date courante via une communication RMI.
    return "non disponible";
  }

  public boolean handleEvent(Event event) {
    // Lorsque le bouton de rafraîchissement est pressé, mettre à jour
    // l'affichage. Utilise les événements du JDK 1.0 pour une portabilité
    // maximale
    switch (event.id) {
      case Event.ACTION_EVENT:
        if (event.target == refresh) {
          refresh();
          return true;
        }
    }
    return false;
  }
}
```

Pour que cette applet puisse être téléchargée par le navigateur client, elle doit être placée sous la racine des documents du serveur, accompagnée d'un document HTML qui y fait référence. Le code HTML pourrait ressembler à celui-ci :

```
<HTML>
<HEAD><TITLE>Applet Daytime</TITLE></HEAD>
<BODY>
<CENTER><H1>Applet Daytime</H1></CENTER>
<CENTER><APPLET CODE=DaytimeApplet CODEBASE=/ WIDTH=300 HEIGHT=180>
</APPLET></CENTER>
</BODY></HTML>
```

Le paramètre CODEBASE indique le répertoire (du point de vue du client) dans lequel le fichier de classe de l'applet a été placé. Un CODEBASE de valeur / indique que les fichiers de classe seront pris dans la racine des documents de l'application web par défaut. Si CODEBASE n'est spécifié, sa valeur par défaut est le répertoire où le fichier HTML a été trouvé. Si l'on suppose que le fichier HTML a été nommé *daytime.html*, cette applet peut être vue à l'URL *http://serveur:port/daytime.html*, et la classe de l'applet sera prise dans *http://serveur:port/DaytimeApplet.class*.

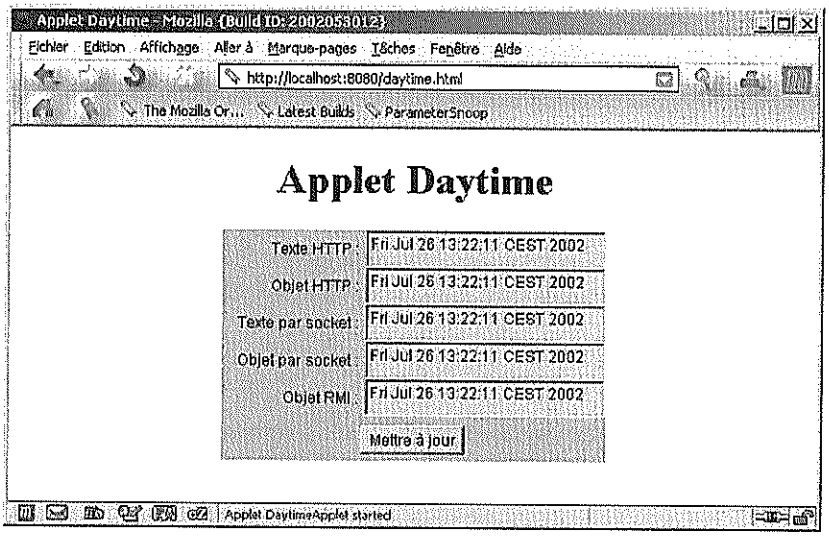


Figure 10-1. L'interface utilisateur de DaytimeApplet

Communication HTTP à base de texte

Commençons par implémenter l'approche commune : la communication HTTP à base de texte.

La servlet

Pour que `DaytimeApplet` puisse retrouver l'heure courante à partir du serveur, elle doit communiquer avec une servlet qui retourne cette heure. *L'exemple 10-2* montre une telle servlet. Elle répond aux requêtes GET et POST par une représentation textuelle de l'heure courante.

Exemple 10-2. DaytimeServlet supportant l'accès HTTP de base

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DaytimeServlet extends HttpServlet {

  public Date getDate() {
    return new Date();
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    out.println(getDate().toString());
  }
```

Exemple 10-2. DaytimeServlet supportant l'accès HTTP de base (suite)

```
  public void doPost(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    doGet(req, res);
  }
}
```

Les fichiers de classes de cette servlet doivent être mis dans l'emplacement standard pour les servlets, habituellement *racine_contexte/WEB-INF/classes*. Une fois que c'est fait, n'importe quel navigateur web peut y accéder par l'URL *http://serveur:port/servlet/DaytimeServlet*.

Retour à l'applet

Maintenant, pour que notre DaytimeApplet accède à cette servlet, elle doit se comporter comme un navigateur et créer une connexion HTTP avec l'URL de la servlet, comme le montre l'implémentation de getDateUsingHttpText() à l'*exemple 10-3*.

Exemple 10-3. DaytimeApplet obtenant l'heure en utilisant HTTP

```
import java.net.URL;                        // Nouvel ajout.
import com.oreilly.servlet.HttpMessage;  // Classe utilitaire.

  private String getDateUsingHttpText() {
    try {
      // Construire une URL qui fait référence à la servlet.
      URL url = new URL(getCodeBase(), "/servlet/DaytimeServlet");

      // Créer un com.oreilly.servlet.HttpMessage pour communiquer avec
      // cette URL.
      HttpMessage msg = new HttpMessage(url);

      // Envoyer un message GET à la servlet, sans chaîne d'interrogation.
      // Obtenir la réponse dans un InputStream.
      InputStream in = msg.sendGetMessage();

      // Envelopper l'InputStream dans un DataInputStream.
      DataInputStream result =
        new DataInputStream(new BufferedInputStream(in));

      // Lire la première ligne de la réponse, qui devrait être une
      // représentation sous forme de chaîne de l'heure courante.
      String date = result.readLine();

      // Fermer l'InputStream.
      in.close();

      // Retourner la date retrouvée.
      return date;
    }
```

Exemple 10-3. DaytimeApplet obtenant l'heure en utilisant HTTP (suite)

```
    catch (Exception e) {
      // S'il y a eu un problème, l'afficher sur System.out (typiquement la
      // console Java) et retourner null.
      e.printStackTrace();
      return null;
    }
  }
```

Cette méthode retrouve l'heure courante sur le serveur en utilisant une connexion HTTP à base de texte. Tout d'abord, elle crée un objet URL qui fait référence à la DaytimeServlet s'exécutant sur le serveur. La machine et le port du serveur pour cette URL viennent de la méthode getCodeBase() de l'applet. Cela garantit qu'ils correspondent à la machine et au port à partir desquels l'applet a été téléchargée. Ensuite, la méthode crée un objet HttpMessage pour communiquer avec cette URL. Cet objet fait tout le travail de bas niveau de création de la connexion. L'applet lui demande de faire une requête GET sur la DaytimeServlet et ensuite elle lit la réponse à partir du InputStream retourné.

Le code de HttpMessage est montré à l'*exemple 10-4*. Il est inspiré de la classe ServletMessage écrite par Rod McChesney de Sun Microsystems.

Exemple 10-4. La classe utilitaire HttpMessage

```
package com.oreilly.servlet;

import java.io.*;
import java.net.*;
import java.util.*;

public class HttpMessage {

  URL servlet = null;
  Hashtable headers = null;

  public HttpMessage(URL servlet) {
    this.servlet = servlet;
  }

  public InputStream sendGetMessage() throws IOException {
    return sendGetMessage(null);
  }

  public InputStream sendGetMessage(Properties args) throws IOException {
    String argString = "";  // Par défaut.

    if (args != null) {
      argString = "?" + toEncodedString(args);
    }
    URL url = new URL(servlet.toExternalForm() + argString);
```

Exemple 10-4. La classe utilitaire HttpMessage (suite)

```
    // Désactiver le cache.
    URLConnection con = url.openConnection();
    con.setUseCaches(false);

    // Envoyer les en-têtes.
    sendHeaders(con);

    return con.getInputStream();
  }

  public InputStream sendPostMessage() throws IOException {
    return sendPostMessage(null);
  }

  public InputStream sendPostMessage(Properties args) throws IOException {
    String argString = "";  // Par défaut.
    if (args != null) {
      argString = toEncodedString(args);  // Notez qu'il n'y a pas de "?".
    }

    URLConnection con = servlet.openConnection();

    // Se préparer pour les entrées et les sorties.
    con.setDoInput(true);
    con.setDoOutput(true);

    // Désactiver le cache.
    con.setUseCaches(false);

    // Contourner un bogue de Netscape.
    con.setRequestProperty("Content-Type",
                           "application/x-www-form-urlencoded");

    // Envoyer les en-têtes.
    sendHeaders(con);

    // Écrire les arguments sous forme de données POST.
    DataOutputStream out = new DataOutputStream(con.getOutputStream());
    out.writeBytes(argString);
    out.flush();
    out.close();

    return con.getInputStream();
  }

  public InputStream sendPostMessage(Serializable obj) throws IOException {
    URLConnection con = servlet.openConnection();
```

Exemple 10-4. La classe utilitaire HttpMessage (suite)

```
    // Se préparer pour les entrées et les sorties.
    con.setDoInput(true);
    con.setDoOutput(true);

    // Désactiver le cache.
    con.setUseCaches(false);

    // Mettre le type de contenu à application/x-java-serialized-object.
    con.setRequestProperty("Content-Type",
                           "application/x-java-serialized-object");

    // Envoyer les en-têtes.
    sendHeaders(con);

    // Écrire l'objet sérialisé sous forme de données POST.
    ObjectOutputStream out = new ObjectOutputStream(con.getOutputStream());
    out.writeObject(obj);
    out.flush();
    out.close();

    return con.getInputStream();
  }

  public void setHeader(String name, String value) {
    if (headers == null) {
      headers = new Hashtable();
    }
    headers.put(name, value);
  }

  // Envoyer au serveur le contenu de la table de hachage des en-têtes.
  private void sendHeaders(URLConnection con) {
    if (headers != null) {
      Enumeration enum = headers.keys();
      while (enum.hasMoreElements()) {
        String name = (String) enum.nextElement();
        String value = (String) headers.get(name);
        con.setRequestProperty(name, value);
      }
    }
  }

  public void setCookie(String name, String value) {
    if (headers == null) {
      headers = new Hashtable();
    }
    String existingCookies = (String) headers.get("Cookie");
    if (existingCookies == null) {
```

Exemple 10-4. La classe utilitaire HttpMessage (suite)

```
      setHeader("Cookie", name + "=" + value);
    }
    else {
      setHeader("Cookie", existingCookies + "; " + name + "=" + value);
    }
  }

  public void setAuthorization(String name, String password) {
    String authorization = Base64Encoder.encode(name + ":" + password);
    setHeader("Authorization", "Basic " + authorization);
  }

  private String toEncodedString(Properties args) {
    StringBuffer buf = new StringBuffer();
    Enumeration names = args.propertyNames();
    while (names.hasMoreElements()) {
      String name = (String) names.nextElement();
      String value = args.getProperty(name);
      buf.append(URLEncoder.encode(name) + "=" + URLEncoder.encode(value));
      if (names.hasMoreElements()) buf.append("&");
    }
    return buf.toString();
  }
}
```

Certains d'entre vous auraient pu s'attendre à ce que la classe `HttpMessage` établisse une connexion par socket avec le serveur et continue en parlant HTTP. Cette approche fonctionnerait certainement, mais ce n'est pas nécessaire. Les classes de plus haut niveau `java.net.URL` et `java.net.URLConnection` fournissent déjà cette fonctionnalité *via* une abstraction pratique.

Examinons rapidement `HttpMessage`. Elle est conçue pour communiquer avec une seule URL, celle passée à son constructeur. Elle peut envoyer de multiples requêtes GET et/ou POST à cette URL, mais elle ne communique qu'avec cette seule URL.

Le code utilisé par `HttpMessage` pour envoyer un message GET est assez simple. Tout d'abord `sendGetMessage()` crée une chaîne d'interrogation d'URL à partir de la liste `java.util.Properties` passée. Elle ajoute cette chaîne d'interrogation à l'URL sauvée, créant un nouvel objet `URL`. À ce stade, on pourrait choisir d'utiliser cette nouvelle URL (de nom `url`) pour communiquer avec la servlet. Un appel à `url.openStream()` retournerait un `InputStream` contenant la réponse. Mais, malheureusement pour nous, par défaut toutes les connexions faites avec un objet `URL` sont mises en cache. Ce n'est pas ce que nous voulons ; nous voulons l'heure courante, pas l'heure de la dernière requête. `HttpMessage` doit donc désactiver le cache[3].

3. Nous pourrions en fait laisser la servlet désactiver le cache, en mettant l'en-tête `Pragma` à `no-cache`. Mais ça ne peut pas faire de mal de l'avoir aussi dans l'applet.

La classe URL ne supporte pas directement ce contrôle de bas niveau, HttpMessage obtient donc l'URLConnection de l'objet URL et lui indique de ne pas utiliser le cache. Enfin, HttpMessage retourne l'InputStream de l'objet URLConnection, qui contient la réponse de la servlet.

Le code utilisé par HttpMessage pour envoyer une requête POST (sendPostMessage()) est similaire. La différence principale est qu'il écrit directement l'information de paramètre codée en URL dans le corps de la requête. Cela correspond au protocole de soumission des informations dans les requêtes POST. L'autre différence est que HttpMessage définit manuellement le type de contenu de la requête à application/x-www-form-urlencoded. Cela permet d'informer le serveur que le contenu POST contient des informations de paramètres.

Nous devons mentionner que HttpMessage est une classe générale pour la communication HTTP. Elle n'est pas forcément utilisée par des applets et elle ne se connecte pas forcément à des servlets. Elle peut être utilisée par tout client Java qui a besoin de se connecter à une ressource HTTP. Elle fait partie du package com.oreilly.servlet car elle est souvent utile pour les communications applet-servlet.

Pour que la classe HttpMessage puisse être utilisée par des applets, elle doit être rendue disponible pour le téléchargement au côté des classes de l'applet. Cela signifie qu'elle doit être mise au bon endroit sous la racine des documents du serveur web. Pour le serveur Tomcat, cet emplacement c'est *racine_serveur/webapps/ROOT/com/oreilly/servlet*. Nous vous recommandons d'y recopier la classe à partir de l'endroit où vous avez installé le paquetage com.oreilly.servlet (probablement *racine_serveur/classes/com/oreilly/servlet*).

Notez que HttpMessage dispose de plusieurs méthodes set permettant de définir les en-têtes de requêtes, les cookies et les informations d'authentification de base destinés au serveur. La méthode setHeader(String name, String value) ajoute un en-tête à la requête ayant le nom et la valeur donnés. Si l'en-tête a déjà été défini, la nouvelle valeur remplace l'ancienne. La méthode setCookie(String name, String value) ajoute un cookie de requête ayant le nom et la valeur donnés. Cette méthode est particulièrement utile pour contourner une limitation dans le suivi de session pour les applets s'exécutant dans le Java Plug-In, comme expliqué au chapitre 7, *Suivi de session*. Enfin, la méthode setAuthorization(String name, String password) ajoute les certificats d'utilisateurs indiqués à la requête, permettant à l'applet d'accéder aux pages protégées par un mot de passe. Elle utilise la classe com.oreilly.servlet.Base64Encoder non montrée dans ce livre mais disponible à *http://www.servlets.com*. Pour toutes ces méthodes, les définitions persistent entre les requêtes et l'appelant doit garantir que le nom et la valeur ne contiennent aucun caractère illégal.

Pour des communications sûres dans une applet, vous pouvez passer à HttpMessage une URL qui commence par https. C'est aussi facile que cela ! (N'oubliez pas que le serveur doit être compatible HTTPS.) Cependant, cela ne fonctionne que pour les applets car le navigateur exécutant l'applet peut faciliter la négociation de la connexion HTTPS. Pour un client non-applet, le constructeur d'URL échoue, indiquant que le protocole HTTPS est inconnu. Vous pouvez contourner ce problème en utilisant la classe com.oreilly.servlet.HttpsMessage disponible à *http://www.servlets.com*. Elle inclut son propre support de HTTPS apporté par Matt Towers. Consultez *http://www.javaworld.com/javaworld/javatips/jw-javatip96.html* pour un article JavaWorld décrivant son comportement.

Maintenant, avec tout ce code fonctionnant ensemble, nous avons une applet qui retrouve l'heure courante à partir d'un serveur en utilisant une communication applet-servlet HTTP à base de texte. Si vous l'essayez, vous devriez voir la date « Texte HTTP » remplie, tandis que les autres dates sont encore marquées « non disponible ».

Communication HTTP à base d'objet

Avec quelques modifications, `DaytimeApplet` peut recevoir la date courante sous forme d'un objet `Date` sérialisé.

La servlet

Pour une compatibilité ascendante, changeons notre `DaytimeServlet` afin qu'elle retourne un `Date` sérialisé uniquement si la requête le demande en passant un paramètre `format` de valeur `object`. Le code est donné à l'*exemple 10-5*.

Exemple 10-5. DaytimeServlet utilisant HTTP pour retourner un objet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DaytimeServlet extends HttpServlet {

  public Date getDate() {
    return new Date();
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // Si le client indique "format=object" alors retourner le Date
    // comme un objet sérialisé
    if ("object".equals(req.getParameter("format"))) {
      ObjectOutputStream out = new ObjectOutputStream(res.getOutputStream());
      out.writeObject(getDate());
    }
    // Sinon, envoyer le Date dans une chaîne normale.
    else {
      PrintWriter out = res.getWriter();
      out.println(getDate().toString());
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    doGet(req, res);
  }
}
```

Comme le montre le code, l'envoi d'un objet Java sérialisé est plutôt simple. Cette technique peut être utilisée pour envoyer tout type Java primitif et/ou tout objet Java qui implémente l'interface `Serializable`, y compris un `Vector` contenant des objets `Serializable`. De multiple objets peuvent aussi être écrits dans le même `ObjectOutputStream`, tant que la classe recevant les objets les lit dans le même ordre et les convertit dans le même type.

Vous pourrez noter que la servlet ne définit pas le type du contenu de la réponse pour indiquer qu'elle contient un objet Java sérialisé. La raison est qu'il n'y pas encore de type MIME standard pour représenter les objets sérialisés. En fait, ce n'est pas vraiment important. Un type de contenu n'est qu'une indication pour le client afin qu'il sache comment traiter ou afficher la réponse. Si une applet suppose qu'elle reçoit un objet Java sérialisé, tout fonctionne bien. Cependant, il est parfois utile d'utilise un type MIME personnalisé (spécifique à votre application), afin que la servlet puisse indiquer à l'applet le contenu de sa réponse.

L'applet

Le code de l'applet pour retrouver l'objet `Date` sérialisé est très similaire au code pour retrouver du texte. *L'exemple 10-6* montre la méthode `getDateUsingHttpObject()`.

Exemple 10-6. DaytimeApplet utilisant HTTP pour retrouver un objet

```
private String getDateUsingHttpObject() {
  try {
    // Construire une URL faisant référence à la servlet.
    URL url = new URL(getCodeBase(), "/servlet/DaytimeServlet");

    // Créer un com.oreilly.servlet.HttpMessage pour communiquer avec cette
    // URL.
    HttpMessage msg = new HttpMessage(url);

    // Construire une liste de propriétés pour dire format=object.
    Properties props = new Properties();
    props.put("format", "object");

    // Envoyer un message GET à la servlet, lui passant "props" dans la
    // chaîne d'interrogation. Obtenir la réponse dans un ObjectInputStream.
    InputStream in = msg.sendGetMessage(props);
    ObjectInputStream result = new ObjectInputStream(in);

    // Lire l'objet Date sur le flot.
    Object obj = result.readObject();
    Date date = (Date)obj;

    // Retourner la représentation chaîne de la Date.
    return date.toString();
  }
  catch (Exception e) {
    // S'il y a eu un problème, l'afficher sur System.out (typiquement la
    // console Java) et retourner null.
```

Exemple 10-6. DaytimeApplet utilisant HTTP pour retrouver un objet (suite)

```
    e.printStackTrace();
    return null;
  }
}
```

Il existe deux différences entre cette méthode et la méthode `getDateUsingHttpText()`. Premièrement, cette méthode crée une liste de `Properties` pour mettre le paramètre `format` à la valeur `object`. Cela indique à `DaytimeServlet` de retourner un objet sérialisé. Deuxièmement, la nouvelle méthode lit le contenu retourné comme un `Object`, en utilisant un `ObjectInputStream` et sa méthode `readObject()`.

Si la classe en train d'être sérialisée ne fait pas partie de l'API du noyau Java (et n'est donc pas déjà disponible pour l'applet), elle doit aussi être rendue disponible au bon endroit sous la racine des documents du serveur. Une applet peut toujours recevoir un contenu d'objet sérialisé, mais elle doit télécharger le fichier de classe correspondant pour reconstruire l'objet.

L'applet peut maintenant retrouver l'heure courante en utilisant une communication HTTP à base de texte ou à base d'objet. Si vous l'essayez (avec un navigateur web ou un visualiseur d'applet qui supportent le JDK 1.1), les champs « Texte HTTP » et « Objet HTTP » devraient être tous deux remplis.

Poster un objet sérialisé ou un fichier

Avant d'aller plus loin, nous devons examiner une autre méthode (jusqu'ici non mentionnée) de la classe `HttpMessage` : `sendPostMessage(Serializable)`. Cette méthode aide une applet à décharger un objet sérialisé sur sa servlet en utilisant la méthode POST. Ce transfert d'objet n'est pas particulièrement utile pour notre exemple (et est un peu hors de propos ici), mais nous le mentionnons car il peut devenir pratique lorsqu'une applet a besoin d'envoyer des structures de données complexes à son serveur. Par exemple, une applet peut décharger un objet `Date`, un objet `Person` ou un tableau `HighScore` contenant des objets `Person` et `Date`. Une applet peut même décharger un document XML, comme un document JDOM (*http://jdom.org*) ou un fichier brut. Rappelez-vous simplement que lorsque vous envoyez un fichier, vous devez envoyer le contenu comme un `byte[]` et non un objet `File` car un objet `File` ne contient que le nom du fichier pas son contenu. L'*exemple 10-7* montre le code de cette méthode.

Exemple 10-7. Poster un objet sérialisé

```
// Décharger un objet sérialisé avec une requête POST. Mettre le type
// de contenu à application/x-java-serialized-object.

public InputStream sendPostMessage(Serializable obj) throws IOException {
  URLConnection con = servlet.openConnection();

  // Se préparer aux entrées et aux sorties.
  con.setDoInput(true);
  con.setDoOutput(true);
```

Exemple 10-7. Poster un objet sérialisé (suite)

```
  // Désactiver le cache.
  con.setUseCaches(false);

  // Mettre le type de contenu à application/x-java-serialized-object.
  con.setRequestProperty("Content-Type",
                         "application/x-java-serialized-object");

  // Écrire l'objet sérialisé comme données POST.
  ObjectOutputStream out = new ObjectOutputStream(con.getOutputStream());
  out.writeObject(obj);
  out.flush();
  out.close();

  return con.getInputStream();
}
```

Une applet utilise `sendPostMessage(Serializable)` comme elle utilise `sendPostMessage(Properties)`. Voici le code d'une applet qui envoie à une servlet les exceptions qu'elle rencontre :[4]

```
catch (Exception e) {
  URL url = new URL(getCodeBase(), "/servlet/ExceptionLogger");
  HttpMessage msg = new HttpMessage(url);
  InputStream in = msg.sendPostMessage(e);
}
```

La servlet reçoit l'`Exception` dans sa méthode `doPost()` de la façon suivante :

```
ObjectInputStream objin = new ObjectInputStream(req.getInputStream());
Object obj = objin.readObject();
Exception e = (Exception) obj;
```

La servlet peut recevoir le type de l'objet envoyé dans le sous-type (seconde moitié) du type de contenu. Notez que cette méthode `sendPostMessage(Serializable)` ne décharge qu'un objet à la fois et uniquement des objets sérialisables (c'est-à-dire pas les types primitifs).

Communication par socket

Regardons maintenant comment une applet et une servlet peuvent communiquer en utilisant une connexion par socket non-HTTP.

4. Normalement, quand une applet lance une exception non gérée, le navigateur journalise la trace de la pile sur sa console Java. Ce code permet à l'applet de lancer l'exception vers le serveur où un journal de toutes les traces de pile des exceptions peuvent être stockées et visualisées par le créateur de l'applet.

La servlet

Le rôle de la servlet dans cette technique de communication est celui d'un auditeur passif. Du fait des restrictions de sécurité, seule l'applet peut initier une connexion par socket. Une servlet doit se contenter d'écouter sur un port de socket et attendre qu'une applet se connecte. De façon générale, une servlet doit commencer à attendre les connexions des applets dans sa méthode `init()` et arrêter son écoute dans sa méthode `destroy()`. Entre les deux, pour chaque connexion qu'elle reçoit, elle doit démarrer un thread de gestion qui communique avec le client.

Dans les connexions par sockets HTTP, ces détails sont pris en charge par le serveur web. Le serveur attend les requêtes HTTP entrantes et les distribue de façon appropriée en appelant les méthodes `service()`, `doGet()` ou `doPost()` d'une servlet. Mais quand une servlet n'opte pas pour une communication HTTP, le serveur web n'est d'aucune aide. La servlet agit, par essence, comme son propre serveur et doit donc gérer elle-même les connexions par sockets.

Bon, peut-être vous a-t-on fait paniquer plus que de raison. La vérité est que l'on peut écrire une classe mère de servlet qui cache les détails de la gestion des connexions par socket. Cette classe, que nous appelons `DaemonHttpServlet`, peut être étendue par toute servlet souhaitant se rendre disponible *via* une communication par socket non-HTTP.[5]

`DaemonHttpServlet` commence par attendre les requêtes du client dans sa méthode `init()` et arrête d'écouter dans sa méthode `destroy()`. Entre les deux, pour chaque connexion reçue, elle appelle la méthode abstraite `handleClient(Socket)`. Cette méthode est implémentée par toute servlet qui étend `DaemonHttpServlet`.

L'*exemple 10-8* montre comment `DaytimeServlet` étend `DaemonHttpServlet` et implémente `handleClient()` afin de se rendre disponible *via* une communication par socket non-HTTP.

Exemple 10-8. Serveur HTTP

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.DaemonHttpServlet;

public class DaytimeServlet extends DaemonHttpServlet {

  public Date getDate() {
    return new Date();
  }
```

5. Le nom *daemon* a été choisi pour faire référence aux démons d'Unix, ces programmes qui s'exécutent silencieusement en arrière-plan gérant certains événements. Et d'où ces programmes ont-ils pris le nom de « daemon » ? Si l'on en croit le *New Hacker's Dictionary*, cela vient « de son sens mythologique, (mais était) plus tard rationalisé comme l'acronyme de 'Disk And Execution MONitor' ».

Exemple 10-8. Serveur HTTP (suite)

```
  public void init() throws ServletException {
    // Comme précédemment, il n'est pas nécessaire d'invoquer super.init()
    // dans la méthode d'initialisation sans argument, mais il faut appeler
    // super.init(config) à partir de l'ancienne méthode init(ServletConfig).
    // Voir le chapitre 3 pour les détails.
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // Si le client indique "format=object" alors envoyer le Date
    // comme un objet sérialisé.
    if ("object".equals(req.getParameter("format"))) {
      ObjectOutputStream out = new ObjectOutputStream(res.getOutputStream());
      out.writeObject(getDate());
    }
    // Sinon envoyer le Date comme une chaîne ASCII normale.
    else {
      PrintWriter out = res.getWriter();
      out.println(getDate().toString());
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    doGet(req, res);
  }

  public void destroy() {
    // Maintenant, pas comme avant, si l'on surcharge destroy(), il faut
    // aussi appeler super.destroy().
    super.destroy();
  }

  // Gérer une connexion par socket du client en démarrant un thread
  // DaytimeConnection.
  public void handleClient(Socket client) {
    new DaytimeConnection(this, client).start();
  }
}

class DaytimeConnection extends Thread {

  DaytimeServlet servlet;
  Socket client;

  DaytimeConnection(DaytimeServlet servlet, Socket client) {
    this.servlet = servlet;
    this.client = client;
```

Exemple 10-8. Serveur HTTP (suite)

```
    setPriority(NORM_PRIORITY - 1);
  }

  public void run() {
    try {
      // Lire la première ligne envoyée par le client, comme du texte Latin-1.
      BufferedReader in = new BufferedReader(
                          new InputStreamReader(
                          client.getInputStream(), "ISO-8859-1"));
      String line = in.readLine();

      // Si c'était "object" alors envoyer le Date comme un objet sérialisé.
      if ("object".equals(line)) {
        ObjectOutputStream out =
          new ObjectOutputStream(client.getOutputStream());
        out.writeObject(servlet.getDate());
        out.close();
      }
      // Sinon, envoyer le Date comme une chaîne normale.
      else {
        // Envelopper un PrintStream autour du OutputStream de la Socket.
        PrintStream out = new PrintStream(client.getOutputStream());
        out.println(servlet.getDate().toString());
        out.close();
      }

      // S'assurer de fermer la connexion.
      client.close();
    }
    catch (IOException e) {
      servlet.log("IOException pendant la gestion de la requête client", e);
    }
    catch (Exception e) {
      servlet.log("Exception pendant la gestion de la requête client);
    }
  }
}
```

La classe `DaytimeServlet` reste en grande partie inchangée. La différence principale est qu'elle étend `DaemonHttpServlet` et implémente une méthode `handleClient(Socket)` qui démarre un nouveau thread `DaytimeConnection`. Cette instance de `DaytimeConnection` prend en charge le traitement d'une connexion par socket spécifique.

`DaytimeConnection` fonctionne de la façon suivante. À sa création elle sauve une référence au `DaytimeServlet`, de manière à pouvoir appeler sa méthode `getDate()`, et une référence au `Socket`, de manière à pouvoir communiquer avec le client. `DaytimeConnection` met aussi sa priorité d'exécution à un en-dessous de la priorité normale, pour indiquer que cette communication peut attendre si nécessaire tandis que d'autres threads effectuent un travail plus critique.

Immédiatement après avoir créé le thread DaytimeConnection, DaytimeServlet démarre le thread, provoquant l'appel de la méthode run(). Dans cette méthode, le DaytimeConnection communique avec le client en utilisant un protocole sans nom (mais absolument pas HTTP). Elle commence par lire la première ligne envoyée par le client. Si cette ligne est object, elle retourne l'heure courante sous forme d'un objet Date sérialisé. Sinon, elle retourne l'heure courante sous forme d'une chaîne de caractères normale. Quand c'est fini, elle ferme la connexion.

La classe mère

La gestion de bas niveau de la socket est réalisée par la classe DaemonHttpServlet. En général, cette classe peut être utilisée sans modification, mais il est utile d'en comprendre le fonctionnement. Le code est montré à l'*exemple 10-9*.

Exemple 10-9. La classe mère DaemonHttpServlet

```
package com.oreilly.servlet;

import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public abstract class DaemonHttpServlet extends HttpServlet {

  protected int DEFAULT_PORT = 1313;  // Non static ni final.
  private Thread daemonThread;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);

    // Démarrer un thread démon.
    try {
      daemonThread = new Daemon(this);
      daemonThread.start();
    }
    catch (Exception e) {
      getServletContext().log(e,
         "Problème de démarrage du thread démon de la socket serveur");
    }
  }

  // Retourner le port de la socket sur lequel cette servlet va écouter.
  // Une servlet peut spécifier le port de trois manières différentes : avec
  // le paramètre d'initialisation socketPort, en définissant la variable
  // DEFAULT_PORT avant d'appeler super.init() ou en surchargeant
  // l'implémentation de cette méthode.
  protected int getSocketPort() {
    try { return Integer.parseInt(getInitParameter("socketPort")); }
```

Exemple 10-9. La classe mère DaemonHttpServlet (suite)

```
    catch (NumberFormatException e) { return DEFAULT_PORT; }
  }

  abstract public void handleClient(Socket client);

  public void destroy() {
    // Stopper le thread démon.
    try {
      daemonThread.stop();
      daemonThread = null;
    } catch (Exception e) {
      getServletContext().log(e,
         "Problème d'arrêt du thread démon de la socket serveur");
    }
  }
}

// Ce travail est découpé en une classe d'aide afin que cette sous-classe de
// DaemonHttpServlet puisse définir sa propre méthode run() sans problème.

class Daemon extends Thread {

  private ServerSocket serverSocket;
  private DaemonHttpServlet servlet;

  public Daemon(DaemonHttpServlet servlet) {
    this.servlet = servlet;
  }

  public void run() {
    try {
      // Créer une socket serveur pour accepter des connexions.
      serverSocket = new ServerSocket(servlet.getSocketPort());
    } catch (Exception e) {
      servlet.getServletContext().log(e,
         "Problème pour établir la socket serveur");
      return;
    }

    try {
      while (true) {
        // Chaque fois qu'une connexion arrive, appeler handleClient().
        // Notez que cette méthode est bloquante. La servlet est responsable
        // de créer un thread de traitement pour des connexions longues.
        try {
          servlet.handleClient(serverSocket.accept());
        } catch (IOException ioe) {
          servlet.getServletContext().log(ioe,
            "Problème pour accepter une connexion par socket du client");
```

Exemple 10-9. La classe mère DaemonHttpServlet (suite)

```
        }
      }
    }
    catch (ThreadDeath e) {
      // Quand le thread est tué, fermer la socket du serveur.
      try {
        serverSocket.close();
      } catch (IOException ioe) {
        servlet.getServletContext().log(ioe,
           "Problème de fermeture de la socket serveur");
      }
    }
  }
}
```

La méthode `init()` de `DaemonHttpServlet` crée et démarre un nouveau thread `Daemon` en charge d'attendre des connexions entrantes. La méthode `destroy()` stoppe le thread. Il est donc impératif que toute servlet étendant `DaemonHttpServlet` appelle `super.init(config)` et `super.destroy()` si la servlet implémente ses propres méthodes `init()` et `destroy()`. (Une servlet écrite conformément à l'API Servlet 2.1 et implémentant la nouvelle méthode `init()` sans argument n'a pas besoin d'appeler `super.init()`, mais elle doit appeler `super.destroy()`.)

Le thread `Daemon` commence par établir un `ServerSocket` qui écoute sur un port de socket spécifique. Ce port est déterminé par un appel à la méthode `getSocketPort()` de la servlet. La valeur retournée est soit la valeur du paramètre d'initialisation `socketPort`, soit, si ce paramètre n'existe pas, la valeur courante de la variable `DEFAULT_PORT`. Une servlet peut décider de surcharger l'implémentation de `getSocketPort()` si elle le souhaite.

Après avoir établi le `ServerSocket`, le thread `Daemon` attend des requêtes entrantes par un appel à `serverSocket.accept()`. Cette méthode est bloquante, elle arrête l'exécution du thread jusqu'à ce qu'un client s'attache à la socket du serveur. À ce moment-là, la méthode `accept()` retourne un objet `Socket` que le thread `Daemon` passe immédiatement à la méthode `handleClient()` de la servlet. Cette méthode `handleClient()` démarre généralement un thread de gestion et retourne immédiatement, laissant le thread `Daemon` prêt à accepter une autre connexion.

Le nettoyage de la socket du serveur est tout aussi important que son établissement. Nous devons être certains que la socket du serveur vit aussi longtemps que la servlet, mais pas plus. Pour cela, la méthode `destroy()` de `DaemonHttpServlet` appelle la méthode `stop()` du thread `Daemon`. Cependant, cet appel n'arrête pas immédiatement le thread `Daemon`. Il provoque juste une exception `ThreadDeath` lancée dans le thread `Daemon` à son point d'exécution courant. Le thread `Daemon` intercepte cette exception et ferme la socket du serveur.

Nous devons faire deux mises en garde lors de l'écriture d'une servlet qui agit comme un serveur non-HTTP. Premièrement, une seule servlet à la fois peut écouter sur un port de socket particulier. Il est donc vital que chaque servlet démon choisisse son propre port de socket, en définissant son paramètre d'initialisation `socketPort`, en définissant la variable `DEFAULT_PORT` avant d'appeler `super.init(config)` ou en surchargeant

directement `getSocketPort()`. Deuxièmement, une servlet démon doit être chargée dans son serveur et sa méthode `init()` doit avoir été appelée avant qu'elle ne puisse accepter des connexions non-HTTP entrantes. Ainsi, vous devez, soit indiquer à votre serveur de la charger au démarrage, soit être sûr qu'elle est toujours accédée *via* HTTP avant d'y accéder directement.

L'applet

Le code de l'applet pour se connecter à la servlet en utilisant une communication non-HTTP, principalement les méthodes `getDateUsingSocketText()` et `getDateUsingSocketObject()`, est montré à l'*exemple 10-10*.

Exemple 10-10. DaytimeApplet obtenant l'heure via une connexion par socket

```
import java.net.Socket;                    // Nouvel ajout.

static final int DEFAULT_PORT = 1313;    // Nouvel ajout.

private int getSocketPort() {
  try { return Integer.parseInt(getParameter("socketPort")); }
  catch (NumberFormatException e) { return DEFAULT_PORT; }
}

private String getDateUsingSocketText() {
  Socket socket = null;
  try {
    // Établir une connexion par socket avec la servlet.
    socket = new Socket(getCodeBase().getHost(), getSocketPort());

    // Imprimer une ligne blanche, indiquant que nous voulons l'heure
    // sous forme de texte.
    PrintStream out = new PrintStream(socket.getOutputStream());
    out.print("\r\n");  // Puisque le comportement de println() varie
                        // avec le système.
    out.flush();

    // Lire la première ligne de la réponse. Elle doit contenir l'heure
    // courante.
    InputStream in = socket.getInputStream();
    DataInputStream result =
      new DataInputStream(new BufferedInputStream(in));
    String date = result.readLine();

    // Retourner la chaîne retrouvée.
    return date;
  } catch (Exception e) {
    // S'il y a eu un problème, l'afficher sur System.out (typiquement la
    // console Java) et retourner null.
    e.printStackTrace();
    return null;
  }
```

Exemple 10-10. DaytimeApplet obtenant l'heure via une connexion par socket (suite)

```
    finally {
      // Toujours fermer la connexion. Ce code s'exécute quelle que soit la
      // façon dont le bloc try se termine.
      if (socket != null) {
        try { socket.close(); }
        catch (IOException ignored) { }
      }
    }
  }

  private String getDateUsingSocketObject() {
    Socket socket = null;
    try {
      // Établir une connexion par socket avec la servlet.
      Socket socket = new Socket(getCodeBase().getHost(), getSocketPort());

      // Imprimer une ligne disant "object", indiquant que nous voulons
      // l'heure sous forme d'un objet Date sérialisé.
      PrintStream out = new PrintStream(socket.getOutputStream());
      out.print("object\r\n");  // Puisque le comportement de println() varie
                               // avec le système.
      out.flush();

      // Créer un ObjectInputStream pour lire la réponse.
      InputStream in = socket.getInputStream();
      ObjectInputStream result =
        new ObjectInputStream(new BufferedInputStream(in));

      // Lire un objet et le convertir en Date.
      Object obj = result.readObject();
      Date date = (Date)obj;

      // Retourner une représentation chaîne de la Date retrouvée.
      return date.toString();
    } catch (Exception e) {
      // S'il y a eu un problème, l'afficher sur System.out (typiquement la
      // console Java) et retourner null.
      e.printStackTrace();
      return null;
    }
    finally {
      // Toujours fermer la connexion. Ce code s'exécute quelle que soit la
      // façon dont le bloc try se termine.
      if (socket != null) {
        try { socket.close(); }
        catch (IOException ignored) { }
      }
    }
  }
```

Pour ces deux méthodes, l'applet commence par créer un Socket utilisé pour communiquer avec la servlet. Pour cela, elle doit connaître le nom de machine et le numéro de port sur lequel la servlet écoute. Déterminer la machine est facile ; il s'agit de celle à partir de laquelle l'applet a été téléchargée, accessible par un appel à `getCodeBase().getHost()`. Pour le port, c'est plus difficile car il dépend entièrement de la servlet à laquelle l'applet va se connecter. Cette applet utilise la méthode `getSocketPort()` pour le déterminer. L'implémentation de `getSocketPort()` montrée ici retourne la valeur du paramètre `socketPort` de l'applet, ou (si ce paramètre n'est pas donné) retourne la valeur de la variable `DEFAULT_PORT`.

Une fois qu'elle a établi une connexion par socket, l'applet suit un protocole sans nom pour communiquer avec la servlet. Ce protocole impose que l'applet envoie une ligne pour indiquer si elle veut l'heure courante sous forme de texte ou d'objet. Si la ligne indique `object`, elle reçoit un objet. Sinon elle reçoit un texte. Après avoir envoyé cette ligne, l'applet peut lire la réponse de façon appropriée.

L'applet et la servlet pourraient continuer à communiquer en utilisant la socket. C'est l'un des principaux avantages de ne pas utiliser une communication HTTP. Mais, dans ce cas, l'applet a eu ce qu'elle voulait et ferme simplement la connexion. Cette fermeture se fait dans un bloc `finally`. Cela garantit que la connexion est bien fermée, qu'une exception soit lancée ou non.

Avec ces deux méthodes, notre applet est presque complète. Si vous l'exécutez maintenant, vous devriez voir que tous les champs, sauf « Objet RMI », contiennent une date.

Communication RMI

Nous avons souligné précédemment dans ce chapitre que l'une des raisons de ne pas utiliser la communication RMI est sa complexité. Bien que ce soit vrai, il est aussi vrai qu'avec l'aide d'une autre classe mère de servlet, le code pour que la servlet devienne accessible *via* une communication RMI peut être ridiculement simple. Tout d'abord, nous vous guiderons pas à pas dans les étapes permettant de faire d'une servlet un objet distant. Ensuite, après avoir vu combien c'est simple et facile, nous vous expliquerons ce qui se passe en coulisses.

La servlet

Pour commencer, tous les objets RMI distants doivent implémenter une interface spécifique. Cette interface fait deux choses : elle déclare les méthodes de l'objet distant qui seront accessibles par les clients distants et elle étend l'interface `Remote` pour indiquer qu'elle est l'interface d'un objet distant. Pour notre `DaytimeServlet`, nous pouvons écrire l'interface `DaytimeServer` montrée à l'*exemple 10-11*.

Exemple 10-11. L'interface DaytimeServer

```
import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface DaytimeServer extends Remote {
  public Date getDate() throws RemoteException;
}
```

Cette interface déclare que notre DaytimeServlet rend la méthode getDate() accessible par les clients distants. Notez que la signature de getDate() a été légèrement modifiée, elle lance maintenant une RemoteException. Chaque méthode rendue accessible *via* RMI doit déclarer qu'elle lance cette exception. Bien que la méthode elle-même puisse ne pas lancer l'exception, celle-ci peut l'être par le système pour indiquer un problème de service réseau.

Le code de DaytimeServlet est pratiquement inchangé. En fait, les seules modifications sont qu'il implémente maintenant DaytimeServer et étend com.oreilly.servlet.RemoteHttpServlet, la classe mère qui permet de ne pas modifier le code de cette servlet. La servlet implémente aussi une méthode destroy() qui appelle super.destroy(). Il est vrai que cette méthode est parfaitement inutile dans cet exemple, mais elle montre que chaque méthode destroy() implémentée dans une servlet distante doit appeler super.destroy() pour donner une chance à la méthode destroy() de l'objet RemoteHttpServlet de terminer la communication RMI. L'*exemple 10-12* montre le nouveau code de DaytimeServlet.

Exemple 10-12. DaytimeServlet supportant maintenant l'accès RMI

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.RemoteHttpServlet;          // Nouvel ajout.

public class DaytimeServlet extends RemoteHttpServlet  // Nouvel ajout.
                            implements DaytimeServer { // Nouvel ajout.

  // La seule méthode de DaytimeServer.
  // Note : la clause throws n'est pas nécessaire ici.
  public Date getDate() {
    return new Date();
  }

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    // Le code supplémentaire vient ici.
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException {
    // Si le client dit "format=object" alors envoyer le Date
    // sous forme d'objet sérialisé.
    if ("object".equals(req.getParameter("format"))) {
      ObjectOutputStream out = new ObjectOutputStream(res.getOutputStream());
      out.writeObject(getDate());
    }
    // Sinon, envoyer le Date comme une chaîne ASCII normale.
```

Exemple 10-12. DaytimeServlet supportant maintenant l'accès RMI (suite)

```
    else {
      PrintWriter out = res.getWriter();
      out.println(getDate().toString());
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    doGet(req, res);
  }

  public void destroy() {
    // Si vous surchargez destroy() vous devez appeler super.destroy().
    super.destroy();
  }
}
```

Et voilà comment l'on écrit une servlet sous forme d'objet distant. La compilation d'un objet distant est identique à celle des autres objets, avec toutefois une étape supplémentaire. Après avoir compilé le code source de la servlet, vous devez compiler la classe de la servlet avec le compilateur RMI *rmic*. Ce compilateur prend un fichier de classe d'un objet distant et génère des versions stub et squelette de la classe. Ces classes fonctionnent en coulisses pour permettre la communication RMI. Vous n'avez pas à vous inquiéter des détails, mais vous devez savoir que le stub sert à l'invocation par le client des méthodes de l'objet distant et que le squelette sert au traitement de ces invocations par le serveur.

L'utilisation de *rmic* est similaire à celle de *javac*. Par exemple, vous pouvez compiler `DaytimeServlet` avec la commande suivante :

```
% rmic DaytimeServlet
```

Notez que vous donnez à *rmic* un nom de classe Java, pas un fichier. Ainsi, si la servlet à compiler fait partie d'un paquetage, il faut la passer à *rmic* sous la forme *nom.paquetage.NomServlet*. Le programme *rmic* peut prendre un chemin de classes *via* le paramètre `-classpath`, ainsi qu'un répertoire de destination pour les fichiers stub et squelette avec le paramètre `-d`.

Après avoir exécutée la commande *rmic* précédente, vous devez voir deux nouveaux fichiers de classe : `DaytimeServlet_Stub.class` et `DaytimeServlet_Skel.class`. Nous vous dirons quoi en faire dans une minute. Tout d'abord vous devez savoir qu'il n'est pas nécessaire d'exécuter le compilateur RMI chaque fois que vous modifiez le code de la servlet distante. Tout simplement parce que les classes du stub et du squelette sont construites à partir de l'interface de la servlet, pas de l'implémentation de cette interface. En revanche, vous devez les regénérer lorsque vous modifiez l'interface de `DaytimeServer` (ou de votre interface équivalente).

Passons maintenant à la dernière étape de l'écriture d'une servlet distante : copier quelques fichiers de classe dans la racine des documents du serveur, d'où ils pourront être téléchargés par une applet. Deux fichiers doivent être téléchargés : la classe du stub

`DaytimeServlet_Stub.class` et la classe d'interface distante `DaytimeServer.class`. Le client (dans ce cas l'applet) a besoin de la classe du stub pour effectuer sa part de la communication RMI, et la classe du stub a elle-même besoin de la classe d'interface distante. Attention, la servlet a aussi besoin de ces fichiers, copiez-les donc dans la racine des documents du serveur et mettez-les dans le chemin de classes du serveur[6]. La *figure 10-2* montre où vont tous les fichiers du serveur.

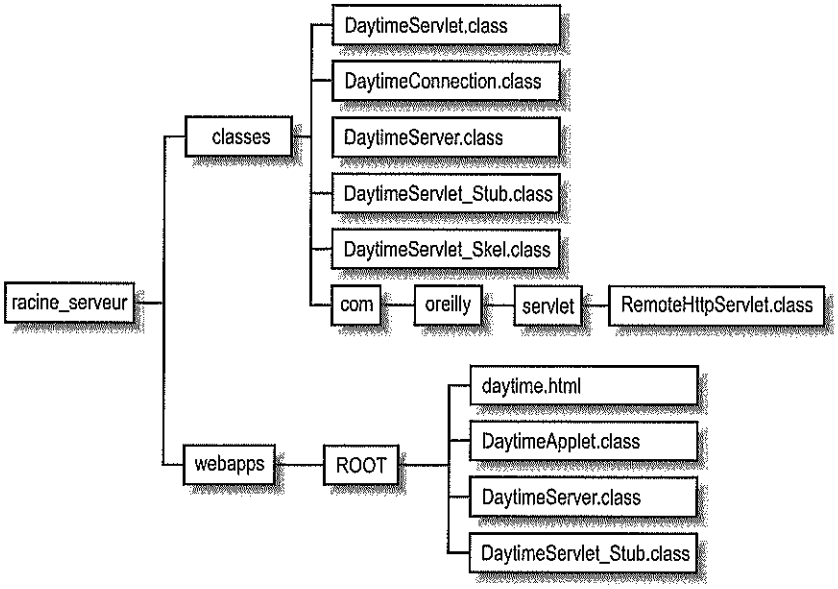


Figure 10-2. Emplacement des fichiers pour la communication RMI

Et voilà ! Si vous avez bien suivi les instructions vous devez obtenir une servlet distante qui fonctionne. Maintenant examinons la classe `RemoteHttpServlet` et voyons ce qui se passe en coulisses.

La classe mère

Un objet distant a besoin de deux choses pour se préparer à une communication RMI : il doit s'exporter et s'enregistrer. Lorsqu'un objet distant s'exporte, il commence par attendre sur un port l'arrivée de requêtes d'invocation de méthodes. Lorsqu'un objet distant s'enregistre, il indique son nom et son numéro de port à un serveur registre, afin que les clients puissent le localiser (essentiellement trouver son numéro de port) et

6. La gestion de multiples fichiers de classes peut devenir un véritable casse-tête pendant le développement. Sur un système Unix, vous pouvez utiliser les liens symboliques pour simplifier cette tâche. Ou, sur n'importe quel système, vous pouvez implémenter une solution générique : changer le chemin des classes du serveur pour y inclure *racine_serveur/webapps/ROOT/classes*. (Notez que ce n'est pas *WEB-INF/classes*.) Mettez-y la classe du stub et la classe d'interface. Le serveur pourra ensuite les retrouver dans son nouveau chemin de classes et l'applet pourra les retrouver elle aussi en mettant `codebase` à */classes*.

communiquer avec lui. Ces deux tâches sont réalisées par la classe RemoteHttpServlet montrée à l'*exemple 10-13*.

Exemple 10-13. La classe mère RemoteHttpServlet

```
package com.oreilly.servlet;

import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public abstract class RemoteHttpServlet extends HttpServlet
                                        implements Remote {

  protected Registry registry;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
      // S'exporter.
      UnicastRemoteObject.exportObject(this);
      // S'enregistrer.
      bind();
    } catch (RemoteException e) {
      getServletContext().log(e, "Problème de liaison au registre RMI");
    }
  }

  public void destroy() {
    // Se désenregistrer.
    unbind();
  }

  // Retourner le nom sous lequel on s'est enregistré.
  protected String getRegistryName() {
    // Le premier choix de nom est le paramètre d'initialisation
    // "registryName".
    String name = getInitParameter("registryName");
    if (name != null) return name;

    // Le choix de repli est le nom de cette classe.
    return this.getClass().getName();
  }

  // Retourner le port sur lequel le serveur de registre va écouter
  // (ou devrait écouter).
```

Exemple 10-13. La classe mère RemoteHttpServlet (suite)

```
  protected int getRegistryPort() {
    // Le premier choix du port est le paramètre "registryPort".
    try { return Integer.parseInt(getInitParameter("registryPort")); }

    // Le choix de repli est le port par défaut (1099).
    catch (NumberFormatException e) { return Registry.REGISTRY_PORT; }
  }

  protected void bind() {
    // Essayer de trouver le registre adéquat en cours d'exécution.
    try {
      registry = LocateRegistry.getRegistry(getRegistryPort());
      registry.list();  // Vérifier qu'il est bien vivant.
    } catch (Exception e) {
      // Impossible d'obtenir un registre valide.
      registry = null;
    }

    // Si l'on ne peut pas en trouver, nous devons le créer.
    // (Équivalent à exécuter "rmiregistry".)
    if (registry == null) {
      try {
        registry = LocateRegistry.createRegistry(getRegistryPort());
      } catch (Exception e) {
        log("Impossible d'obtenir ou de créer un registre RMI sur le port " +
            getRegistryPort() + ": " + e.getMessage());
        return;
      }
    }

    // Si nous arrivons ici c'est que nous avons un registre valide.
    // Maintenant enregistrer cette instance de servlet auprès de ce registre.
    // "Rebind" afin de remplacer tout autre objet en utilisant notre nom.
    try {
      registry.rebind(getRegistryName(), this);
    } catch (Exception e) {
      log("Impossible de se lier au registre RMI : " + e.getMessage());
      return;
    }
  }

  protected void unbind() {
    try {
      if (registry != null) registry.unbind(getRegistryName());
    } catch (Exception e) {
      getServletContext().log(e, "Problème pour quitter le registre RMI");
    }
  }
}
```

Si vous avez déjà utilisé RMI ou lu à son sujet, vous avez probablement vu des objets distants qui étendent la classe `java.rmi.server.UnicastRemoteObject`. C'est la façon standard, et en fait celle recommandée, d'écrire un objet distant. Cependant, la classe `RemoteHttpServlet` n'étend pas `UnicastRemoteObject` ; elle étend `HttpServlet`. Comme vous devez le savoir, Java ne supporte pas l'héritage multiple. Cela signifie que `RemoteHttpServlet` doit choisir entre étendre soit `UnicastRemoteObject` soit `HttpServlet`, même si elle a besoin des fonctionnalités des deux. Le choix est difficile. Quelle que soit la classe que `RemoteHttpServlet` n'étend pas, elle doit réimplémenter sa fonction elle-même. Nous avons choisi d'étendre `HttpServlet` car il est plus facile de réécrire la fonction de `UnicastRemoteObject` que celle de `HttpServlet`.

Cette réécriture fait que `RemoteHttpServlet` réalise deux choses qu'elle n'aurait pas eu à faire si elle avait étendu `UnicastRemoteObject`. Premièrement, elle doit déclarer qu'elle implémente l'interface `Remote`. Tous les objets distants doivent implémenter cette interface, mais normalement en étendant `UnicastRemoteObject` une classe l'obtient sans effort. Le prix de l'indépendance n'est pas encore trop élevé, l'interface `Remote` ne définissant aucune méthode. Un objet déclare qu'il implémente `Remote` uniquement pour exprimer son souhait d'être traité comme un objet distant.

La seconde chose que doit faire `RemoteHttpServlet` est de s'exporter manuellement. Normalement, c'est fait automatiquement dans le constructeur `UnicastRemoteObject()`. Mais une fois encore, ce n'est pas vraiment un problème. La classe `UnicastRemoteObject` a une méthode statique `exportObject(Remote)` que tout objet `Remote` peut utiliser pour s'exporter lui-même. `RemoteHttpServlet` utilise cette méthode et s'exporte en une seule ligne :

```
UnicastRemoteObject.exportObject(this);
```

Ces deux étapes, implémenter `Remote` et s'exporter, sont faites par `RemoteHttpServlet` au lieu d'étendre `UnicastRemoteObject`[7].

Le reste du code de `RemoteHttpServlet` comprend son enregistrement et son désenregistrement auprès d'un registre RMI. Comme nous l'avons dit précédemment, un serveur registre RMI sert de lieu à partir duquel les clients peuvent localiser les objets serveur. Un objet distant (objet serveur) s'enregistre lui-même auprès du registre sous un certain nom. Les clients consultent ensuite le registre pour rechercher l'objet par son nom. Pour se rendre accessible par les clients, notre servlet doit trouver (ou créer) un serveur registre et s'enregistrer auprès de ce serveur sous un nom spécifique. Cette action s'appelle aussi la *liaison au registre*. `RemoteHttpServlet` effectue cette liaison avec sa méthode `bind()`, appelée dans sa méthode `init()`.

---

7. Pour être absolument correct, il y a autre chose à faire. D'après la documentation de `java.rmi.remote.UnicastRemoteObject`, « Si `UnicastRemoteObject` n'est pas étendu, la classe doit prendre la responsabilité des méthodes `hashCode`, `equals` et `toString` héritées de la classe `Object`, afin que leur comportement soit correct pour les objets distants. ». D'après la documentation de `java.rmi.remote.RemoteRef`, « Ces méthodes doivent garantir que deux stubs d'objet distant qui font référence au même objet distant auront le même code de hachage (de façon à pouvoir utiliser les objets distants comme clé dans des tables de hachage) ». L'implémentation du mécanisme permettant de garantir cela est plutôt complexe et, pensons-nous, généralement pas nécessaire pour la communication applet-servlet ; nous avons donc décidé d'éluder cette responsabilité dans `RemoteHttpServlet`.

Où exécuter le registre ?

La manière communément admise pour exécuter un serveur registre RMI passe par le programme Java autonome *rmiregistry*. Cependant, nous vous recommandons de ne pas utiliser *rmiregistry* mais plutôt de laisser `RemoteHttpServlet` créer le registre elle-même. C'est plus facile et plus efficace. La première servlet qui a besoin du registre peut le créer. Et, en démarrant le registre dans une servlet, le registre s'exécute dans la même JVM que celle de la servlet. Il est ainsi possible de n'utiliser qu'une seule JVM pour le serveur, toutes ses servlets (objets distants) et le registre. En fait, certains serveurs d'applications lancent automatiquement un registre dans leur propre JVM à leur démarrage.

Il est aussi possible de ne pas lancer de registre du tout, en utilisant les servlets. Un client peut se connecter à une servlet spéciale *via* HTTP et demander un objet distant donné (à l'aide d'un paramètre). La servlet retourne dans ses données de réponse un stub sérialisé pour cet objet distant ; exactement ce qu'aurait renvoyé le registre. En utilisant une servlet à la place de *rmiregistry*, il est possible d'implémenter une politique de sécurité sur les clients qui demandent des objets, de laisser la servlet donner à chaque client une référence à un objet distant différent qu'elle peut choisir, et (puisque la servlet peut construire l'objet distant dans le traitement de la requête) permettre à l'objet distant d'être disponible avant même qu'il soit créé.

La méthode `bind()` utilise deux méthodes support, `getRegistryPort()` et `getRegistryName()`, pour déterminer le port sur lequel la servlet doit s'exécuter et le nom sous lequel elle doit s'enregistrer. Dans les implémentations actuelles, le port est pris à partir du paramètre d'initialisation `registryPort`, ou vaut 1099 par défaut. Le nom est pris dans le paramètre d'initialisation `registryName` ou est égal au nom de la classe de la servlet par défaut, dans ce cas `DaytimeServlet`.

Examinons pas à pas la méthode `bind()`. Elle commence par utiliser le code suivant pour trouver un registre adéquat déjà en cours d'exécution :

```
registry = LocateRegistry.getRegistry(getRegistryPort());
registry.list();
```

La première ligne essaie d'obtenir le registre s'exécutant sur le port donné. La seconde demande au registre de lister les objets enregistrés. Si les deux appels réussissent, nous avons un registre valide. Si l'un des appels lance une `Exception`, la méthode `bind()` détermine qu'il n'existe pas de registre valide et en crée un elle-même :

```
registry = LocateRegistry.createRegistry(getRegistryPort());
```

Après cela, la méthode `bind()` doit, soit avoir trouvé, soit avoir créé un serveur registre. Si elle échoue, elle retourne et la servlet n'est pas enregistrée. `RemoteHttpServlet` se lie ensuite au registre en utilisant la ligne de code :

```
registry.rebind(getRegistryName(), this);
```

Elle utilise la méthode `Registry.rebind()` au lieu de la méthode `Registry.bind()` pour indiquer que cette liaison doit remplacer toute liaison précédente de même nom. La liaison persiste jusqu'à ce que la servlet soit détruite, instant auquel la méthode de-

stroy() de RemoteHttpServlet appelle sa méthode unbind(). Le code utilisé par unbind() pour se délier du registre est remarquablement simple :

```
if (registry != null) registry.unbind(getRegistryName());
```

Il demande simplement au registre de délier son nom.

Notez qu'une servlet distante doit être chargée dans son serveur et sa méthode init() doit avoir été appelée avant qu'elle ne soit prête pour la communication RMI. Ainsi, comme avec une servlet démon, vous devez, soit indiquer à votre serveur de la charger au démarrage, soit être sûr qu'on y accède toujours *via* HTTP avant d'être accédée directement.

L'applet

Passons maintenant du serveur au client. Le code utilisé par notre DaytimeApplet pour invoquer la méthode getDate() de notre nouvelle DaytimeServlet est montré à l'*exemple 10-14*.

Exemple 10-14. DaytimeApplet obtenant l'heure en utilisant RMI

```
import java.rmi.*;           // Nouvel ajout.
import java.rmi.registry.*;  // Nouvel ajout.

private String getRegistryHost() {
  return getCodeBase().getHost();
}

private int getRegistryPort() {
  try { return Integer.parseInt(getParameter("registryPort")); }
  catch (NumberFormatException e) { return Registry.REGISTRY_PORT; }
}

private String getRegistryName() {
  String name = getParameter("registryName");
  if (name == null) {
    name = "DaytimeServlet";  // Par défaut.
  }
  return name;
}

private String getDateUsingRMIObject() {
  try {
    Registry registry =
      LocateRegistry.getRegistry(getRegistryHost(), getRegistryPort());
    DaytimeServer daytime =
      (DaytimeServer)registry.lookup(getRegistryName());
    return daytime.getDate().toString();
  }
  catch (ClassCastException e) {
    System.out.println("L'objet retrouvé n'est pas un DaytimeServer : " +
                       e.getMessage());
  }
```

Exemple 10-14. DaytimeApplet obtenant l'heure en utilisant RMI (suite)

```
  catch (NotBoundException e) {
    System.out.println(getRegistryName() + " non lié : " + e.getMessage());
  }
  catch (RemoteException e) {
    System.out.println("Exception distante : " + e.getMessage());
  }
  catch (Exception e) {
    System.out.println(
        "Problème pour obtenir une référence à DaytimeServer : " +
        e.getClass().getName() + " : " + e.getMessage());
  }
  return null;
}
```

Les trois premières méthodes sont des méthodes de support. `getRegistryHost()` retourne la machine sur laquelle le serveur registre doit s'exécuter. Ce doit toujours être la machine à partir de laquelle l'applet a été téléchargée. `getRegistryPort()` retourne le port sur lequel le serveur registre doit écouter. C'est normalement le port de registre par défaut 1099, bien qu'il puisse être changé par le paramètre `registryPort`. `getRegistryName()` retourne le nom sous lequel la servlet doit s'être enregistrée. Par défaut c'est `DaytimeServlet`, mais il peut être changé par le paramètre `registryName`.

La véritable recherche de l'objet servlet distant et l'invocation de sa méthode `getDate()` ont lieu dans ces trois lignes de la méthode `getDateUsingRMIObject()` :

```
Registry registry =
  LocateRegistry.getRegistry(getRegistryHost(), getRegistryPort());
DaytimeServer daytime =
  (DaytimeServer)registry.lookup(getRegistryName());
return daytime.getDate().toString();
```

La première ligne localise le registre pour la machine et le port indiqués. La deuxième ligne utilise le registre pour rechercher l'objet distant enregistré sous le nom indiqué et en même temps fait de l'objet un objet `DaytimeServer`. La troisième ligne invoque la méthode `getDate()` de cet objet et reçoit en retour un objet `Date` sérialisé. Ensuite, dans la même ligne, on récupère la représentation du `Date` sous forme d'un `String`.

Le reste de la méthode `getDateUsingRMIObject()` traite les exceptions qui pourraient arriver pendant ces trois lignes. Elle intercepte une `ClassCastException` si l'objet retrouvé n'est pas un `DaytimeServer`, une `NotBoundException` si le registre n'a pas d'objet enregistré sous le nom donné et une `RemoteException` en cas de problème sur le réseau. Elle intercepte aussi une `Exception` générale en cas d'autre problème.

On peut se demander pourquoi `DaytimeApplet` utilise `Registry.lookup(String)` au lieu de `java.rmi.Naming.lookup(String)` pour retrouver sa référence à la servlet distante. En fait, il n'y a pas vraiment de raison ; c'est une simple affaire de goût personnel. On pourrait tout aussi bien remplacer les deux premières lignes de `getDateUsingRMIObject()` par le code suivant :

```
DaytimeServer daytime =
  (DaytimeServer)Naming.lookup("rmi://" + getRegistryHost() +
                          ":" + getRegistryPort() + "/" + getRegistryName());
```

Et voilà pour la cinquième et dernière méthode de DaytimeApplet. Allez-y et exécutez l'applet maintenant. Voyez-vous chaque champ de date rempli ? Vous ne devriez pas. Vous devriez voir des valeurs vides dans les options de communication par socket. Souvenez-vous que nous avons enlevé le support des communications pas socket lorsque nous avons fait de DaytimeServlet un objet distant. Remettons maintenant la communication par socket.

Une servlet complète

Nous avons maintenant besoin d'une seule servlet qui peut se rendre disponible *via* une communication HHTP, une communication par socket non-HTTP et une communication RMI. Une servlet de cette sorte peut étendre une nouvelle classe mère, com.oreilly.servlet.RemoteDaemonHttpServlet, implémentant les possibilités discutées jusqu'ici pour une RemoteHttpServlet et une DaemonHttpServlet.

Voici le code qui déclare cette servlet complète :

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.RemoteDaemonHttpServlet;

public class DaytimeServlet extends RemoteDaemonHttpServlet
                            implements DaytimeServer {

  public Date getDate() {
    return new Date();
  }

  // Le reste ne change pas.
```

Ce code est presque le même qu'à l'*exemple 10-8*. À la base, c'est le même code réécrit afin de déclarer que l'on étend RemoteDaemonHttpServlet et que l'on implémente DaytimeServer.

Le code de la classe mère RemoteDaemonHttpServlet est aussi pratiquement le même que pour RemoteHttpServlet. Il n'y a que deux modifications : il étend DaemonHttpServlet au lieu de HttpServlet et sa méthode destroy() appelle d'abord super.destroy() :

```
package com.oreilly.servlet;

import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public abstract class RemoteDaemonHttpServlet extends DaemonHttpServlet
                                              implements Remote {

  public void destroy() {
    super.destroy();
    unbind();
  }

  // Le reste ne change pas
```

Maintenant, notre `DaytimeApplet` peut se connecter à cette servlet démon revue et produire la sortie complète montrée précédemment à la *figure 10-1*.

Serveur de discussion

L'exemple du serveur horaire de la dernière section a montré les rouages des trois techniques de communication applet-servlet. En revanche, il ne tirait pas parti des gains permanents de l'utilisation de la communication par socket. Il n'a pas non plus montré la simplicité de la communication RMI ou l'élégance des rappels RMI (la servlet peut invoquer les méthodes de l'applet). De même, il n'a pas donné de raison satisfaisante justifiant pourquoi une servlet devrait supporter toutes les techniques de communication ; il n'y avait pas d'état à maintenir ni de code complexe à mettre dans un endroit. C'est pourquoi, avant de terminer notre discussion sur la communication applet-servlet, examinons un exemple plus complexe : un serveur de discussion (*chat*), implémenté comme une servlet supportant les clients qui se connectent *via* HTTP, des sockets non-HTTP et RMI.

Nous construirons ce serveur de discussion en utilisant les trois techniques de communication afin de pouvoir tirer parti de la meilleure solution pour chaque client. Par exemple, lorsque le client supporte RMI, la servlet peut être traitée comme un objet distant, et (quand c'est possible) elle peut traiter l'applet comme un objet distant. Lorsque le client ne supporte pas RMI mais qu'il accepte la communication par socket directe, le serveur de communication peut utiliser la persistance de socket et communiquer avec le client en utilisant un protocole par socket non-HTTP. Et bien sûr, quand tout cela échoue, le serveur de discussion peut se rabattre sur HTTP. Ce n'est pas recommandé car HTTP, protocole sans état, exige de chaque client qu'il demande une mise à jour. Mais pour beaucoup de clients, HTTP est l'unique possibilité.

Le serveur de discussion est implémenté comme une seule classe avec une seule instanciation car l'état qu'il conserve est important et une grande partie de son code devrait sinon être répété. Le diviser en trois classes, une pour chaque protocole, demanderait une communication inter-serveurs importante et une réplication du code principal du serveur dans les trois classes. L'implémentation du serveur de discussion sous forme de servlet est une façon simple de rendre un objet accessible *via* les trois techniques de communication. En étant une servlet HTTP, il intègre le support de HTTP. Et en étendant la classe `RemoteDaemonHttpServlet`, il peut aussi facilement avoir accès aux communications par sockets non-HTTP et RMI.

Notez que bien que vous verrez le code en totalité, nous n'expliquerons pas complètement chaque ligne. Pour cela, il nous faudrait étendre la taille de ce chapitre plus que

de raisonnable, si l'on suppose que ce n'est pas déjà le cas. Ainsi, nous expliquerons les problèmes quand ils concernent la communication applet-servlet et vous laisserons étudier le code pour comprendre tous les détails.

Conception

La *figure 10-3* montre l'applet de discussion en action. Notez qu'elle utilise un grand composant `TextArea` pour afficher la conversation en cours et un petit composant `TextInput` par lequel l'utilisateur peut poster un message d'une seule ligne. Au fur et à mesure que chaque contributeur compose un message, il est envoyé au serveur de discussion et distribué à chaque client de diverses façons.

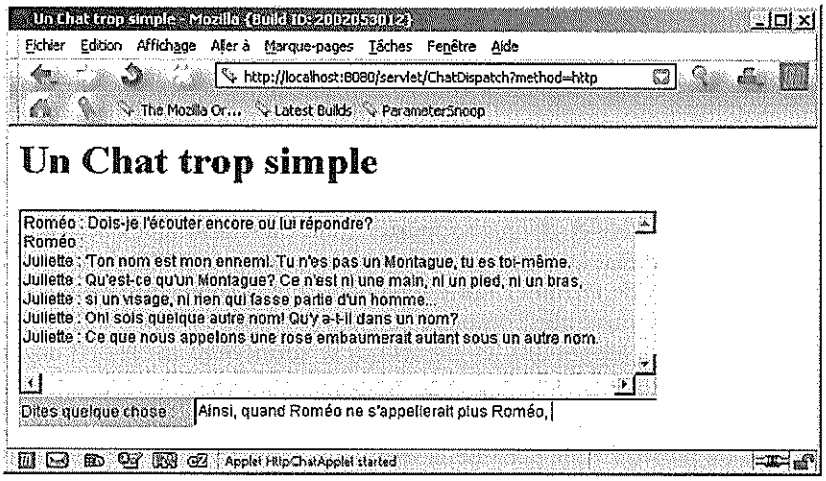


Figure 10-3. L'applet de discussion en action

Les clients de discussion HTTP postent leurs messages dans le serveur en utilisant la méthode POST de HTTP. L'applet prend le nouveau message du composant `TextInput` lorsque l'utilisateur appuie sur Entrée, elle code le message comme une URL et le poste à la servlet comme un paramètre de message. Tout cela est très facile. En revanche, la façon dont chaque client de discussion HTTP récupère les messages des autres clients est un peu plus compliquée. On utilise la méthode GET de HTTP pour recevoir chaque message, mais il y a un problème : le client ne sait pas exactement quand un nouveau message est disponible. C'est le problème du principe de communication unidirectionnelle par requête/réponse. Le client doit, soit regarder périodiquement s'il y a des mises à jour, soit simuler une communication bidirectionnelle en faisant une série de requêtes GET bloquantes. C'est-à-dire que le client de discussion initie une requête GET qui bloque jusqu'à ce que le serveur décide qu'il est temps de retourner quelque chose. Dans notre exemple, nous allons implémenter cette communication bidirectionnelle simulée.

Les clients de discussion par socket, pour des raisons pratiques, postent leurs messages sur le serveur de la même façon que les clients HTTP, par la méthode POST. Ils pourraient les poster en utilisant des connexions par socket, mais le gain en efficacité est marginal alors que, du moins dans ce cas, la complexité augmente considérablement.

Cependant, ces clients utilisent les sockets pour lire les messages des autres clients, remplaçant la communication bidirectionnelle simulée par une communication bidirectionnelle réelle. Dès qu'un nouveau message arrive dans la servlet, celle-ci l'envoie immédiatement aux clients de discussion *via* des connexions par socket de type texte.

Les clients de discussion RMI effectuent leurs POST et leurs GET en utilisant l'invocation de méthodes. Pour poster chaque nouveau message, l'applet appelle simplement la méthode `broadcastMessage(String)` de la servlet distante. Pour récupérer les nouveaux messages, il existe deux options. Le client peut appeler la méthode bloquante `getNextMessage()` de la servlet, ou, *via* les rappels, il peut demander à la servlet d'appeler sa propre méthode `setNextMessage(String)` à chaque nouvelle diffusion de messages. Nous avons choisi d'utiliser l'option des *callbacks* (rappels) dans notre exemple.

Devant toutes ces applets, il y a une applet de distribution. Elle permet au client de choisir sa technique de communication applet-servlet (HTTP, socket ou RMI) et, en fonction de son choix, elle génère une page contenant l'applet appropriée. Il est vrai que l'on pourrait écrire une seule applet pour supporter ces trois techniques et en sélectionner une automatiquement en fonction de l'environnement d'exécution, mais cela compliquerait inutilement notre exemple. La servlet de distribution donne aussi à l'applet le nom de son utilisateur, mais laissons cela pour plus tard.

La servlet

Les listings complets de l'interface `ChatServer` et de la classe `ChatServlet` l'implémentant sont donnés à l'*exemple 10-15* et l'*exemple 10-16*.

Exemple 10-15. L'interface ChatServer, implémentée par ChatServlet

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatServer extends Remote {
  public String getNextMessage() throws RemoteException;
  public void broadcastMessage(String message) throws RemoteException;

  public void addClient(ChatClient client) throws RemoteException;
  public void deleteClient(ChatClient client) throws RemoteException;
}
```

Exemple 10-16. Un service complet de discussion serveur/servlet

```
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.RemoteDaemonHttpServlet;
```

Exemple 10-16. Un service complet de discussion serveur/servlet (suite)

```
public class ChatServlet extends RemoteDaemonHttpServlet
                         implements ChatServer {

  // source agit comme le distributeur de nouveaux messages.
  MessageSource source = new MessageSource();

  // socketClients détient des références vers tous les clients connectés
  // par socket.
  Vector socketClients = new Vector();

  // rmiClients détient des références vers tous les clients RMI.
  Vector rmiClients = new Vector();

  // doGet() retourne le nouveau message. Elle bloque jusqu'à ce qu'il y en
  // ait un.
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    // Retourner le prochain message (bloquant).
    out.println(getNextMessage());
  }

  // doPost() accepte un nouveau message et le diffuse à tous les clients
  // HTTP, par socket et RMI actuellement à l'écoute.
  public void doPost(HttpServletRequest req, HttpServletResponse res)
                               throws ServletException, IOException {
    // Accepter le nouveau message comme paramètre "message".
    String message = req.getParameter("message");

    // Le diffuser à tous les clients écoutant.
    if (message != null) broadcastMessage(message);

    // Mettre le code d'état pour indiquer qu'il n'y aura pas réponse.
    res.setStatus(res.SC_NO_CONTENT);
  }

  // getNextMessage() retourne le prochain nouveau message.
  // Elle bloque tant qu'il n'est pas disponible.
  public String getNextMessage() {
    // Créer un MessageSink pour attendre qu'un nouveau message arrive de
    // la source.
    return new MessageSink().getNextMessage(source);
  }

  // broadcastMessage() informe tous les clients écoutant qu'il y a un
  // nouveau message. Tous les appels à getNextMessage() sont débloqués.
```

Exemple 10-16. Un service complet de discussion serveur/servlet (suite)

```
public void broadcastMessage(String message) {
  // Envoyer le message à tous les clients connectés par HTTP en donnant
  // le message à la source.
  source.sendMessage(message);

  // Envoyer directement le message à tous les clients connectés par
  // socket.
  Enumeration enum = socketClients.elements();
  while (enum.hasMoreElements()) {
    Socket client = null;
    try {
      client = (Socket)enum.nextElement();
      PrintStream out = new PrintStream(client.getOutputStream());
      out.println(message);
    }
    catch (IOException e) {
      // Problème avec un client, le fermer et le supprimer.
      try {
        if (client != null) client.close();
      }
      catch (IOException ignored) { }
      socketClients.removeElement(client);
    }
  }

  // Envoyer directement le message à tous les clients RMI.
  enum = rmiClients.elements();
  while (enum.hasMoreElements()) {
    ChatClient chatClient = null;
    try {
      chatClient = (ChatClient)enum.nextElement();
      chatClient.setNextMessage(message);
    }
    catch (RemoteException e) {
      // Problème de communication avec un client, l'enlever.
      deleteClient(chatClient);
    }
  }
}

protected int getSocketPort() {
  // On écoute sur le port 2428 (regardez un téléphone pour savoir pourquoi).
  return 2428;
}

public void handleClient(Socket client) {
  // On a un nouveau client par socket. L'ajouter à notre liste.
  socketClients.addElement(client);
}
```

Exemple 10-16. Un service complet de discussion serveur/servlet (suite)

```
  public void addClient(ChatClient client) {
    // On a un nouveau client RMI. L'ajouter à notre liste.
    rmiClients.addElement(client);
  }

  public void deleteClient(ChatClient client) {
    // Enlever le client spécifié de notre liste.
    rmiClients.removeElement(client);
  }
}

// MessageSource agit comme la source de nouveaux messages. Les clients
// intéressés par recevoir de nouveaux messages peuvent observer cet objet.
class MessageSource extends Observable {
  public void sendMessage(String message) {
    setChanged();
    notifyObservers(message);
  }
}

// MessageSink agit comme le receveur de nouveaux messages. Il écoute
// la source.
class MessageSink implements Observer {

  String message = null;  // Défini par update() et lu par getNextMessage().

  // Appelée par la source du message lorsqu'on obtient un nouveau message.
  synchronized public void update(Observable o, Object arg) {
    // Obtenir le nouveau message.
    message = (String)arg;

    // Réveiller notre thread bloqué.
    notify();
  }

  // Obtenir le message suivant à partir de la source.
  synchronized public String getNextMessage(MessageSource source) {
    // Indiquer à la source que l'on veut être informé des nouveaux messages.
    source.addObserver(this);

    // Attendre jusqu'à ce que update() reçoive un message.
    while (message == null) {
      try { wait(); } catch (Exception ignored) { }
    }

    // Indiquer à la source de ne plus nous avertir.
    source.deleteObserver(this);
```

Exemple 10-16. Un service complet de discussion serveur/servlet (suite)

```
    // Maintenant, retourner le message reçu. Mais commencer par mettre
    // la variable d'instance de message à null afin que update() et
    // getNextMessage() puissent être appelées à nouveau.
    String messageCopy = message;
    message = null;
    return messageCopy;
  }
}
```

Les méthodes `getNextMessage()` et `broadcastMessage(String message)` sont les parties les plus intéressantes de `ChatServlet`. La méthode `getNextMessage()` retourne le prochain message quand il arrive ; elle bloque jusqu'à ce que cet événement se produise. Pour permettre l'attente, elle utilise les classes `MessageSource` et `MessageSink`. Sans trop entrer dans les détails, nous dirons juste ceci : la servlet construit un nouveau `MessageSink` et lui demande de récupérer le nouveau message à sa source. Pour cela, il s'enregistre lui-même comme observateur de la source et appelle `wait()` pour attendre. Lorsque la source reçoit un nouveau message, il (étant un observateur) est informé du changement par un appel à sa méthode `update()`. Cette méthode sauve le dernier message de la source dans la variable `message` et appelle ensuite `notify()`. Cela provoque le déblocage de la méthode `getNextMessage()` qui retourne le message.

La méthode `broadcastMessage()` indique à tous les clients qui écoutent qu'un nouveau message est disponible. Elle informe les clients HTTP en envoyant le message à `MessageSource` ; les autres clients sont informés directement en bouclant sur la liste des clients. Pour chacun de ses clients connectés par socket, le message est envoyé sur la socket du client. Pour chaque client RMI, elle appelle la méthode `setNextMessage(String)` du client. C'est le rappel dont nous avons parlé. Si, à n'importe quel moment, il y a un problème avec un client par socket ou RMI, ce client est enlevé de la liste.

Les deux listes `socketClients` et `rmiClients` sont remplies lorsque la servlet reçoit un appel des clients. Quand un client par socket se connecte, la méthode `handleClient(Socket)` de la servlet est appelée et le nouveau client est ajouté au vecteur `socketClients`. Les clients RMI s'ajoutent eux-même à la liste en invoquant la méthode `addClient(ChatClient)` de la servlet.

Les méthodes `doGet()` et `doPost()` de `ChatServlet` sont essentiellement de fines enveloppes autour des méthodes `getNextMessage()` et `broadcastMessage()`. `doGet()` est si fine que vous pouvez pratiquement voir au travers : `doGet()` renvoie pour réponse tout `String` retourné par `getNextMessage()`. `doPost()` est un peu moins transparente. Elle extrait le message posté à partir du paramètre `message` venant des données du formulaire POST, diffuse le message en le passant à la méthode `broadcastMessage()` et met le code d'état de sa réponse à `SC_NO_CONTENT` pour indiquer que la réponse ne contient rien. En un sens, effectuer une requête GET est équivalent à appeler `getNextMessage()` et effectuer une requête POST est équivalent à appeler `broadcastMessage()`. Le seul problème est que dans la conception HTTP, il existe un petit délai entre le moment où le client obtient un message et le moment où il en demande un nouveau. Cela peut conduire à des pertes de messages quand la charge est élevée. Une meilleure conception (laissée au lecteur) serait d'utiliser un objet `HttpSession` du client pour contenir une file des messages en attente.

Avez-vous noté le port sur lequel ChatServlet écoute ? C'est 2428. Cette façon de surcharger la méthode getSocketPort() comme le fait ChatServlet permet de définir facilement le numéro de port de la socket lorsque vous ne voulez pas utiliser un paramètre d'initialisation.

L'applet HTTP

Le code de notre première applet, l'applet de discussion HTTP, est montré à l'*exemple 10-17*.

Exemple 10-17. Un client de discussion utilisant la communication HTTP

```
import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;

import com.oreilly.servlet.HttpMessage;

public class HttpChatApplet extends Applet implements Runnable {

  TextArea text;
  Label label;
  TextField input;
  Thread thread;
  String user;

  public void init() {
    // Vérifier si cette applet a été chargée directement à partir du système
    // de fichiers. Si c'est le cas, expliquer à l'utilisateur que cette
    // applet doit être chargée à partir d'un serveur afin qu'elle
    // communique avec les servlets de ce serveur.
    URL codebase = getCodeBase();
    if (!"http".equals(codebase.getProtocol())) {
      System.out.println();
      System.out.println("*** Oups ! ***");
      System.out.println("Cette applet doit être chargée à partir d'un ");
      System.out.println("serveur web. Veuillez recommencer, cette fois en ");
      System.out.println("demandant le fichier HTML contenant cette applet ");
      System.out.println("à \"http://serveur:port/file.html\".");
      System.out.println();
      System.exit(1);  // Ne marche qu'avec appletviewer. Les navigateurs
                       // lancent une exception.
    }

    // Obtenir le nom de cet utilisateur à partir d'un paramètre d'applet
    // défini par la servlet. Nous pourrions simplement le demander à
    // l'utilisateur, mais cela montre une forme de communication
    // servlet->applet.
```

Exemple 10-17. Un client de discussion utilisant la communication HTTP (suite)

```
    user = getParameter("user");
    if (user == null) user = "anonymous";

    // Mettre en place l'interface utilisateur... En haut, un grand TextArea
    // montrant ce que chacun dit. En dessous, un TextField pour accepter
    // l'entrée de l'utilisateur.
    text = new TextArea();
    text.setEditable(false);
    label = new Label("Dites quelque chose : ");
    input = new TextField();
    input.setEditable(true);

    setLayout(new BorderLayout());
    Panel panel = new Panel();
    panel.setLayout(new BorderLayout());

    add("Center", text);
    add("South", panel);

    panel.add("West", label);
    panel.add("Center", input);
  }

  public void start() {
    thread = new Thread(this);
    thread.start();
  }

  String getNextMessage() {
    String nextMessage = null;
    while (nextMessage == null) {
      try {
        URL url = new URL(getCodeBase(), "/servlet/ChatServlet");
        HttpMessage msg = new HttpMessage(url);
        InputStream in = msg.sendGetMessage();
        DataInputStream data = new DataInputStream(
                               new BufferedInputStream(in));
        nextMessage = data.readLine();
      }
      catch (SocketException e) {
        // Connexion à l'hôte impossible, le signaler et attendre
        // avant de réessayer.
        System.out.println("Connexion à l'hôte impossible : " +
            e.getMessage());
        try { Thread.sleep(5000); } catch (InterruptedException ignored) { }
      }
      catch (FileNotFoundException e) {
        // La servlet n'existe pas, le signaler et attendre avant
```

Exemple 10-17. Un client de discussion utilisant la communication HTTP (suite)

```
        // de réessayer.
        System.out.println("Ressource non trouvée : " + e.getMessage());
        try { Thread.sleep(5000); } catch (InterruptedException ignored) { }
      }
      catch (Exception e) {
        // Autres problèmes, le signaler et attendre avant de réessayer.
        System.out.println("Exception générale : " +
          e.getClass().getName() + " : " + e.getMessage());
        try { Thread.sleep(1000); } catch (InterruptedException ignored) { }
      }
    }
    return nextMessage + "\n";
  }

  public void run() {
    while (true) {
      text.appendText(getNextMessage());
    }
  }

  public void stop() {
    thread.stop();
    thread = null;
  }

  void broadcastMessage(String message) {
    message = user + " : " + message;  // Ajouter le nom de l'utilisateur.
    try {
      URL url = new URL(getCodeBase(), "/servlet/ChatServlet");
      HttpMessage msg = new HttpMessage(url);
      Properties props = new Properties();
      props.put("message", message);
      msg.sendPostMessage(props);
    }
    catch (SocketException e) {
      // Connexion à l'hôte impossible, le signaler et abandonner
      // la diffusion.
      System.out.println("Connexion à l'hôte impossible : " + e.getMessage());
    }
    catch (FileNotFoundException e) {
      // La servlet n'existe pas, le signaler et abandonner la diffusion.
      System.out.println("Ressource non trouvée : " + e.getMessage());
    }
    catch (Exception e) {
      // Autres problèmes, le signaler et abandonner la diffusion.
      System.out.println("Exception générale : " +
        e.getClass().getName() + " : " + e.getMessage());
    }
  }
```

Exemple 10-17. Un client de discussion utilisant la communication HTTP (suite)

```
  public boolean handleEvent(Event event) {
    switch (event.id) {
      case Event.ACTION_EVENT:
        if (event.target == input) {
          broadcastMessage(input.getText());
          input.setText("");
          return true;
        }
    }
    return false;
  }
}
```

Cette applet possède les deux mêmes méthodes que `ChatServlet` : `getNextMessage()` et `broadcastMessage()`. Sa méthode `getNextMessage()` retrouve le prochain message à partir de la servlet. Elle est appelée continuellement afin de mettre à jour le `TextArea`. Elle fonctionne en utilisant un `HttpMessage` afin de faire une requête GET sur la servlet, puis d'interpréter la première ligne de la réponse comme étant le nouveau message. Sa méthode `broadcastMessage()` envoie un message à la servlet afin que celle-ci le distribue aux autres clients. Cette méthode est appelée dans la méthode `handleEvent()` de l'applet chaque fois que l'utilisateur appuie sur Entrée dans le composant `TextInput`. Elle fonctionne de façon similaire à `getNextMessage()`. Elle utilise un `HttpMessage` pour faire une requête POST, en passant le texte de `TextInput` dans le paramètre `message`, et elle ne s'occupe pas de la réponse.

L'applet de connexion par socket

La seule différence entre `SocketChatApplet` et `HttpChatApplet` réside dans une méthode `getNextMessage()` revue. Cette méthode est montrée à l'*exemple 10-18*.

Exemple 10-18. Un client de discussion utilisant une connexion par socket

```
static final int PORT = 2428;
DataInputStream serverStream;

String getNextMessage() {
  String nextMessage = null;
  while (nextMessage == null) {
    try {
      // Se connecter au serveur si ce n'est déjà fait.
      if (serverStream == null) {
        Socket s = new Socket(getCodeBase().getHost(), PORT);
        serverStream = new DataInputStream(
                       new BufferedInputStream(
                       s.getInputStream()));
      }

      // Lire une ligne.
```

Exemple 10-18. Un client de discussion utilisant une connexion par socket (suite)

```
      nextMessage = serverStream.readLine();
    }
    catch (SocketException e) {
      // Connexion à l'hôte impossible, le signaler et attendre
      // avant de réessayer.
      System.out.println("Connexion à l'hôte impossible : " + e.getMessage());
      serverStream = null;
      try { Thread.sleep(5000); } catch (InterruptedException ignored) { }
    }
    catch (Exception e) {
      // Autres problèmes, le signaler et attendre avant de réessayer.
      System.out.println("Exception générale : " +
        e.getClass().getName() + " : " + e.getMessage());
      try { Thread.sleep(1000); } catch (InterruptedException ignored) { }
    }
  }
  return nextMessage + "\n";
}
```

Cette méthode lit les messages diffusés à partir de la socket connectée à la servlet de discussion. Elle utilise un protocole simple : tout le contenu est du texte, un message par ligne. La première fois que cette méthode est appelée, elle établit une connexion par socket et l'utilise ensuite pour obtenir un `DataInputStream`, à partir duquel elle peut lire sur la socket une ligne à la fois. Elle lit la première ligne sur ce flot et retourne le texte pour créer le nouveau message. Pour chaque invocation suivante, elle réutilise le même flot et retourne simplement la ligne suivante. Si une `SocketException` survient, elle rétablit la connexion.

L'applet RMI

Le code de l'interface `ChatClient` est montré à l'*exemple 10-19* ; l'applet de discussion à base de RMI qui l'implémente est montrée à l'*exemple 10-20*.

Exemple 10-19. L'interface ChatClient, implémentée par RMIChatApplet

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatClient extends Remote {
  public void setNextMessage(String message) throws RemoteException;
}
```

Exemple 10-20. Un client de discussion utilisant la communication RMI

```
import java.applet.*;
import java.awt.*;
import java.io.*;
```

Exemple 10-20. Un client de discussion utilisant la communication RMI (suite)

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.*;

public class RMIChatApplet extends Applet implements ChatClient {

  TextArea text;
  Label label;
  TextField input;
  Thread thread;
  String user;

  ChatServer chatServer;

  private int getRegistryPort() {
    try { return Integer.parseInt(getParameter("port")); }
    catch (NumberFormatException ignored) { return Registry.REGISTRY_PORT; }
  }

  private String getRegistryName() {
    String name = getParameter("name");
    return (name == null ? "ChatServlet" : name);
  }

  // Retourner une référence au serveur/servlet de discussion distante.
  // Essayer de quitter s'il y a un problème.
  private ChatServer getChatServer() {
    try {
      Registry registry =
        LocateRegistry.getRegistry(getCodeBase().getHost(), getRegistryPort());
      Object obj = registry.lookup(getRegistryName());
      return (ChatServer)obj;
    }
    catch (java.rmi.UnknownHostException e) {
      // Hôte du registre non connu, essayer de quitter.
      System.out.println("Hôte dans l'URL inconnu : " + e.getMessage());
      System.exit(1);
    }
    catch (NotBoundException e) {
      // Impossible de trouver notre objet, essayer de quitter.
      System.out.println("Nom non lié : " + e.getMessage());
      System.exit(1);
    }
    catch (ClassCastException e) {
      // L'objet n'est pas un ChatServer, essayer de quitter.
      System.out.println(getRegistryName() + " n'est pas un ChatServer :" +
                         e.getMessage());
```

Exemple 10-20. Un client de discussion utilisant la communication RMI (suite)

```
      System.exit(1);
    }
    catch (RemoteException e) {
      // Problème général avec RMI, essayer de quitter.
      System.out.println("Exception distante : " + e.getMessage());
      System.exit(1);
    }
    catch (Exception e) {
      // Autre problème, essayer de quitter.
      System.out.println("Exception générale : " +
        e.getClass().getName() + " : " + e.getMessage());
      System.exit(1);
    }
    return null;  // Retourner null si exit() ne marche pas.
  }

  // S'ajouter comme client du serveur de discussion. Notez qu'un registre
  // RMI est inutile.
  private void registerWithChatServer(ChatServer server) {
    try {
      UnicastRemoteObject.exportObject(this);
      server.addClient(this);
    }
    catch (RemoteException e) {
      // Problème général avec RMI, essayer de quitter.
      System.out.println("Exception distante : " + e.getMessage());
      System.exit(1);
    }
    catch (Exception e) {
      // Autre problème, essayer de quitter.
      System.out.println("Exception générale : " +
        e.getClass().getName() + " : " + e.getMessage());
      System.exit(1);
    }
  }

  public void init() {
    // Vérifier si cette applet a été chargée directement à partir du système
    // de fichiers. Si c'est le cas, expliquer à l'utilisateur que cette
    // applet doit être chargée à partir d'un serveur afin qu'elle
    // communique avec les servlets de ce serveur.
    URL codebase = getCodeBase();
    if (!"http".equals(codebase.getProtocol())) {
      System.out.println();
      System.out.println("*** Oups ! ***");
      System.out.println("Cette applet doit être chargée à partir d'un ");
      System.out.println("serveur web. Veuillez recommencer, cette fois en ");
      System.out.println("demandant le fichier HTML contenant cette applet ");
```

Exemple 10-20. Un client de discussion utilisant la communication RMI (suite)

```
      System.out.println("à \"http://serveur:port/file.html\".");
      System.out.println();
      System.exit(1);  // Ne marche qu'avec appletviewer. Les navigateurs
                       // lancent une exception.
    }

    // Obtenir le serveur de discussion distant.
    chatServer = getChatServer();

    // S'enregistrer comme l'un de ses clients.
    registerWithChatServer(chatServer);

    // Obtenir le nom de cet utilisateur à partir d'un paramètre d'applet
    // défini par la servlet de distribution. Nous pourrions simplement le
    // demander à l'utilisateur, mais cela montre une forme de communication
    // servlet->applet.
    user = getParameter("user");
    if (user == null) user = "anonymous";

    // Mettre en place l'interface utilisateur... En haut, un grand TextArea
    // montrant ce que chacun dit. En dessous, un TextField pour accepter
    // l'entrée de l'utilisateur.
    text = new TextArea();
    text.setEditable(false);
    label = new Label("Dites quelque chose : ");
    input = new TextField();
    input.setEditable(true);

    setLayout(new BorderLayout());
    Panel panel = new Panel();
    panel.setLayout(new BorderLayout());

    add("Center", text);
    add("South", panel);

    panel.add("West", label);
    panel.add("Center", input);
  }

  String getNextMessage() {
    String nextMessage = null;
    while (nextMessage == null) {
      try {
        nextMessage = chatServer.getNextMessage();
      }
      catch (RemoteException e) {
        // Exception distante, la signaler et attendre avant de réessayer.
        System.out.println("Exception distante :" + e.getMessage());
```

Exemple 10-20. Un client de discussion utilisant la communication RMI (suite)

```
        try { Thread.sleep(1000); } catch (InterruptedException ignored) { }
      }
    }
    return nextMessage + "\n";
  }

  public void setNextMessage(String message) {
    text.appendText(message + "\n");
  }

  void broadcastMessage(String message) {
    message = user + ": " + message;  // Ajouter le nom de l'utilisateur.
    try {
      chatServer.broadcastMessage(message);
    }
    catch (RemoteException e) {
      // Exception distante, la signaler et abandonner la diffusion.
      System.out.println("Exception distante : " + e.getMessage());
    }
    catch (Exception e) {
      // Autre exception, la signaler et abandonner la diffusion.
      System.out.println("Exception générale : " +
        e.getClass().getName() + " : " + e.getMessage());
    }
  }

  public boolean handleEvent(Event event) {
    switch (event.id) {
      case Event.ACTION_EVENT:
        if (event.target == input) {
          broadcastMessage(input.getText());
          input.setText("");
          return true;
        }
    }
    return false;
  }
}
```

Les implémentations des méthodes `getNextMessage()` et `broadcastMessage()` de cette applet sont aussi simples que possible. Elles ne font qu'appeler les méthodes de même nom de la servlet distante. Mais leur simplicité a un prix : un code de configuration plus complexe. La méthode `init()`, plus particulièrement, doit appeler la longue (mais maintenant compréhensible) méthode `getChatServer()` pour obtenir une référence à la servlet de discussion distante.

Si vous examinez bien `RMIChatApplet`, vous noterez qu'elle n'utilise pas vraiment la méthode `getNextMessage()`. À la place, elle demande à la servlet d'appeler sa méthode `setNextMessage()` chaque fois qu'un nouveau message est diffusé. Cette demande est

faite dans la méthode `init()` par un appel à `registerWithChatServer(ChatServer)`. Cette méthode exporte l'applet sous forme d'objet distant et appelle ensuite la méthode `addClient()` de la servlet en lui passant une référence à elle-même. Après cela, la méthode `broadcastMessage()` de la servlet rappelle l'applet chaque fois qu'il y a un nouveau message.

Si vous essayez d'utiliser vous-même les rappels, n'oubliez pas les bases données précédemment. Vous devez exécuter le compilateur RMI *rmic* sur l'applet distante pour générer ses classes du stub et du squelette. Et vous devez être certain que les fichiers `RMIChatApplet_Stub.class` et `ChatClient.class` sont bien quelque part dans le chemin des classes du serveur.

Il faut aussi faire attention au fait que les navigateurs modernes sont plus restrictifs sur l'autorisation des rappels que ne l'étaient les navigateurs précédents. Le plus souvent, cette possibilité est désactivée par défaut. Quand cela se produit, vous n'obtenez généralement que des messages d'erreur énigmatiques et parfois aucun message d'erreur, juste un échec silencieux. Vous pouvez autoriser les rappels en signant numériquement l'applet (une action différente selon le navigateur) et en accordant aux applets signées des permissions supplémentaires ou en accédant à l'applet en utilisant HTTPS à partir d'un serveur de confiance. De façon générale, on peut voir cela comme un autre inconvénient de RMI et une autre raison de choisir HTTP pour les communications applet-serveur.

Le distributeur

Maintenant voici le dernier exemple de code de ce chapitre, la servlet `ChatDispatch` montrée à l'*exemple 10-21*. Cette servlet effectue deux tâches. Premièrement, lorsqu'on y accède sans paramètre de requête, elle affiche une page de bienvenue accueillante demandant à l'utilisateur quelle version de l'applet il veut utiliser, comme montré à la *figure 10-4*. Deuxièmement, lorsque l'on y accède avec un paramètre de requête, elle affiche une page qui contient l'applet appropriée, comme vous pouvez le voir à la *figure 10-3*. Faites attention que l'URL utilisée pour accéder à cette servlet de distribution contienne bien le vrai nom du serveur et non pas `localhost`, afin d'éviter des problèmes de sécurité RMI.

Exemple 10-21. La servlet de distribution frontale

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ChatDispatch extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws IOException, ServletException {
    res.setContentType("text/html");

    if (!req.getParameterNames().hasMoreElements()) {
      // Pas de paramètres de requête. Afficher une page d'accueil.
      printWelcomePage(req, res);
    }
```

Exemple 10-21. La servlet de distribution frontale (suite)

```
    else {
      // Au moins un paramètre de requête. Afficher une page contenant l'applet.
      printAppletPage(req, res);
    }
  }

  // La page accueille le lecteur et lui propose un formulaire pour
  // qu'il choisisse une méthode de communication applet-servlet.
  private void printWelcomePage(HttpServletRequest req,
                                HttpServletResponse res)
                  throws IOException {
    PrintWriter out = res.getWriter();
    String me = req.getServletPath();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>");
    out.println("Bienvenue sur ce Chat trop simple");
    out.println("</TITLE></HEAD>");
    out.println();
    out.println("<BODY>");
    out.println("<H1>Bienvenue sur ce Chat trop simple</H1>");
    out.println();
    out.println("Voulez-vous communiquer en utilisant :");
    out.println("<UL>");
    out.println("  <LI><A HREF=\"" + me + "?method=http\">http</A>");
    out.println("  <LI><A HREF=\"" + me + "?method=socket\">socket</A>");
    out.println("  <LI><A HREF=\"" + me + "?method=rmi\">rmi</A>");
    out.println("</UL>");
    out.println("</BODY></HTML>");
  }

  // La page affiche l'applet de discussion.
  private void printAppletPage(HttpServletRequest req,
                               HttpServletResponse res)
                  throws IOException {
    PrintWriter out = res.getWriter();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>Un Chat trop simple</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H1>Un Chat trop simple</H1>");

    String method = req.getParameter("method");
    String user = req.getRemoteUser();
    String applet = null;

    if ("http".equals(method)) {
      applet = "HttpChatApplet";
    }
```

Exemple 10-21. La servlet de distribution frontale (suite)

```
    else if ("socket".equals(method)) {
      applet = "SocketChatApplet";
    }
    else if ("rmi".equals(method)) {
      applet = "RMIChatApplet";
    }
    else {
      // Pas de méthode, ou  méthode invalide. Expliquer à l'utilisateur
      // ce que nous attendons.
      out.println("Désolé, cette servlet a besoin d'un paramètre " +
                  "<TT>method</TT> ayant une des valeurs suivantes : " +
                  "http, socket, rmi");
      return;
    }

    // Afficher le code HTML pour générer l'applet. Choisir le code de
    // l'applet en fonction de la méthode. Fournir un paramètre si nous
    // connaissons l'utilisateur distant.
    out.println("<APPLET CODE=" + applet + " CODEBASE=/ " +
                "WIDTH=500 HEIGHT=170>");
    if (user != null)
      out.println("<PARAM NAME=user VALUE=\"" + user + "\">");
    out.println("</APPLET>");

    out.println("</BODY></HTML>");
  }
}
```

Figure 10-4. La page d'accueil pour la discussion

Rien ici ne devrait vous surprendre. En fait, ce code doit vous paraître agréablement simple après l'exemple de `ChatServlet`. Cependant, cet exemple montre une dernière forme de communication applet-servlet : les paramètres d'applet générés par une servlet. Avec cette technique, une servlet génère une page contenant une applet et lui passe des informations en manipulant les balises `<PARAM>`. Toute information que la servlet

veut envoyer à une nouvelle applet peut se faire ainsi. Dans cet exemple, la servlet envoie le nom retourné par `req.getRemoteUser()`. Une servlet pourrait aussi indiquer à l'applet son type de navigateur en lui envoyant la chaîne retournée par `req.getHeader("User-Agent")`. La servlet pourrait également utiliser une base de données pour déterminer les possibilités du navigateur et indiquer à l'applet exactement ce qu'elle doit savoir. Elle pourrait même lui indiquer si le navigateur supporte la communication RMI.

11

Collaboration de servlets

Les servlets qui s'exécutent dans le même serveur ont plusieurs moyens de communiquer les unes avec les autres. Il existe deux principaux styles de collaboration entre les servlets :

Partage d'informations

Cela implique deux servlets ou plus partageant un état ou une ressource. Par exemple, un ensemble de servlets gérant un magasin en ligne pourrait partager les informations sur le stock des produits ou une connexion à la base de données. Le suivi de session (voir le chapitre 7, *Suivi de session*) est un cas spécial de partage d'informations.

Partage du contrôle

Cela implique deux servlets ou plus partageant le contrôle d'une requête. Par exemple, une servlet peut recevoir la requête et laisser à l'autre servlet une partie ou toute la responsabilité du traitement de la requête.

Par le passé (avant l'API Servlet 2.1), nous aurions pu donner un autre style de collaboration : la *manipulation directe*. Dans ce style de collaboration, une servlet obtient une référence directe à une autre servlet *via* la méthode `getServlet()` et en invoque ensuite les méthodes. Cette collaboration n'est plus supportée ; la méthode `getServlet()` a été désapprouvée et elle retourne `null` dans l'API 2.1 et ultérieure. En voici la raison : une servlet pouvant être détruite par le serveur à tout moment, seul le serveur doit détenir une référence à une servlet. Tout ce qu'il est possible de faire avec `getServlet()` peut être mieux fait et de façon plus sûre par les alternatives présentées dans ce chapitre.

Partage d'informations

Les servlets coopèrent souvent en partageant des informations. Ces informations peuvent être un état, une ressource, une fabrique de ressources, ou autre. Dans l'API Servlet 2.0 et précédente, il n'existait aucun mécanisme intégré par lequel les servlets pouvaient partager des informations, et dans la première édition de ce livre (écrit à partir de l'API

2.0) nous avons dû montrer plusieurs solutions créatives, y compris placer des informations dans la liste des propriétés système ! Heureusement, ce type d'astuces n'est plus nécessaire car, à partir de l'API Servlet 2.1, la classe ServletContext a été étendue afin de servir de conteneur d'informations partagées.

Partage avec ServletContext

Une servlet retrouve le ServletContext de son application web par un appel à getServletContext(). Une servlet peut utiliser le contexte comme s'il s'agissait d'un Hashtable ou d'un Map, à l'aide des méthodes suivantes.

```
public void ServletContext.setAttribute(String name, Object o)
public Object ServletContext.getAttribute(String name)
public Enumeration ServletContext.getAttributeNames()
public void ServletContext.removeAttribute(String name)
```

La méthode setAttribute() lie un objet sous le nom indiqué. Toute liaison existante de même nom est remplacée. Les noms d'attributs doivent suivre les conventions de nommage des paquetages pour éviter d'écraser un autre nom. Les noms de paquetage java.*, javax.* et com.sun.* sont réservés.

La méthode getAttribute() retrouve l'objet lié sous le nom indiqué ou null si l'attribut n'existe pas. Cet appel peut aussi retrouver des attributs figés spécifiques au serveur (par exemple javax.servlet.context.tempdir) comme indiqué au chapitre 4, *Retrouver l'information*.

La méthode getAttributeNames() retourne un Enumeration contenant les noms de tous les attributs liés ou un Enumeration s'il n'existe aucune liaison.

La méthode removeAttribute() supprime l'objet lié sous le nom indiqué ou ne fait rien si l'attribut n'existe pas. Il est préférable de supprimer les attributs qui ne sont plus indispensables afin de réduire l'encombrement mémoire.

Utiliser le contexte pour vendre des burritos

Comme exemple amusant, imaginez un ensemble de servlets qui vendent des burritos et partagent une spécialité du jour. Une servlet d'administration peut fixer la spécialité du jour comme le montre l'*exemple 11-1*.

Exemple 11-1. Me gustan burritos

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SpecialSetter extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
```

Exemple 11-1. Me gustan burritos (suite)

```
    ServletContext context = getServletContext();
    context.setAttribute("com.costena.special.burrito", "Pollo Adobado");
    context.setAttribute("com.costena.special.day", new Date());

    out.println("Le burrito spécial a été défini.");
  }
}
```

Ensuite, toute autre servlet du serveur peut accéder à la spécialité et l'afficher à l'aide du code de l'*exemple 11-2*.

Exemple 11-2. Especial del día

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SpecialGetter extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    ServletContext context = getServletContext();
    String burrito = (String)
      context.getAttribute("com.costena.special.burrito");
    Date day = (Date)
      context.getAttribute("com.costena.special.day");

    DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM);
    String today = df.format(day);

    out.println("Aujourd'hui (" + today + "), notre burrito spécial est : " +
                burrito);
  }
}
```

Partage avec un autre ServletContext

L'utilisation de `ServletContext` pour le partage des informations a pour avantage que chaque application web dispose de son propre conteneur d'informations unique. Il n'existe donc pas de risque de collision de noms accidentelle ou même de collision de noms d'une même application déployée deux fois sur un serveur. Cependant, on a parfois besoin de partager les informations entre des contextes web. Dans ce cas, il y a deux solutions. Premièrement, utiliser un conteneur d'informations externe comme un singleton ou une base de données, ou, deuxièmement, utiliser des méthodes spéciales pour

accéder directement à un autre contexte. Nous allons examiner cette deuxième solution.

Une servlet peut obtenir un handle à un autre contexte du même serveur à l'aide de la méthode `getContext()` de son propre contexte :

```
public ServletContext ServletContext.getContext(String uripath)
```

Cette méthode renvoie le `ServletContext` contenant le chemin d'URI indiqué. Le chemin doit être absolu (commençant par /). Il est interprété en partant de la racine des documents du serveur. Cette méthode permet à une servlet d'obtenir l'accès à un contexte différent du sien. Dans un environnement distribué ou sécurisé (voir le chapitre 12, *Servlets d'entreprise et J2EE*), le conteneur de servlet peut retourner `null` pour n'importe quel chemin.

Pour montrer cela, supposez une servlet hors du contexte du magasin de burritos qui veut accéder à la spécialité du jour. De plus, supposez que le contexte du magasin de burritos a pour racine */burritostore*. Et enfin, supposez que le serveur a configuré sa sécurité de manière à autoriser la communication intercontextes. Dans ces conditions, le code suivant permet à toute servlet de retrouver la spécialité du jour à partir de l'application de vente de burritos :

```
ServletContext monContexte = getServletContext();
ServletContext autreContexte =
  monContexte.getContext("/burritostore/index.html");
String burrito = autreContexte.getAttribute("com.costena.special.burrito");
Date jour = (Date)autreContexte.getAttribute("com.costena.special.day");
```

La seule manière d'obtenir une référence à un contexte différent est de rechercher le chemin. Notre exemple ne fonctionnera donc plus si le magasin de burritos change d'URL. Dans le futur, il sera possible d'obtenir un contexte par un nom.

Problèmes avec le chargeur de classes

Si l'objet placé dans le contexte est un objet propre à l'application, son partage entre contextes peut être plus complexe du fait des problèmes liés au chargement des classes. Souvenez-vous que les classes de chaque application se trouvant sous *WEB-INF* sont chargées par un chargeur de classes différent. Rappelez-vous aussi que le chargeur de classes d'une application ne peut localiser les classes stockées dans une autre application. C'est pourquoi un objet dont le fichier *.class* existe uniquement dans *WEB-INF/classes* ou *WEB-INF/lib* n'est pas facilement utilisable par d'autres applications. Toute tentative de transtyper l'objet dans son propre type conduira à une `NoClassDefFoundError`.

La solution consistant à copier le fichier *.class* dans les deux applications ne fonctionne pas non plus. Les classes chargées par différentes chargeurs de classes sont des classes différentes, même si leurs définitions correspondent. Si vous essayez cette solution, vous ne ferez que remplacer `NoClassDefFoundError` par `ClassCastException`.

Il existe trois solutions. Premièrement, copiez le fichier *.class* de l'objet partagé dans le chemin de classes standard du serveur (pour Tomcat, il s'agit de *racine_serveur/classes*). De cette manière l'objet sera trouvé par le chargeur de classes principal partagé par toutes les applications web. Deuxièmement, évitez de transtyper l'objet retourné et invoquez ses méthodes en utilisant l'introspection (une technique par laquelle une classe

Java peut s'inspecter et se manipuler elle-même à l'exécution). Ce n'est pas élégant mais cela fonctionne si vous n'avez pas un plein contrôle sur le serveur. Troisièmement, transtyper l'objet retourné en une interface qui déclare les méthodes pertinentes et placez cette interface dans le chemin de classes standard du serveur. Chaque classe, sauf les interfaces, peut rester dans les applications web individuelles. Cela demande de reconstruire la classe de l'objet partagé mais cela fonctionne bien si l'implémentation doit rester dans les applications web originales.

Partage du contrôle

Pour une collaboration plus dynamique, les servlets peuvent partager le contrôle de la requête. Premièrement, une servlet peut *renvoyer* une requête entière, effectuant un traitement préliminaire et passant ensuite la requête à un autre composant. Deuxièmement, une servlet peut *inclure* dans sa réponse du contenu généré par un autre composant, créant ainsi une forme d'inclusion côté serveur (SSI — *server-side include*). Conceptuellement, si vous pensez à la page résultante comme à un écran, un renvoi donne à une autre servlet un accès total à l'écran, alors qu'une inclusion injecte seulement une partie du contenu dans l'écran à un endroit donné.

Cette possibilité de délégation apporte aux servlets une plus grande souplesse et autorise une meilleure abstraction. En utilisant la délégation, une servlet peut construire sa réponse à l'aide d'un ensemble de contenus générés par divers composants web serveur. Cette fonctionnalité est particulièrement importante pour les JavaServer Pages, où il arrive souvent qu'une servlet prétraite une requête, puis passe la requête à une page JSP pour la fin du traitement (voir le chapitre 18, *JavaServer Pages*).

Obtenir un distributeur de requête

Pour supporter la délégation de requête, l'API Servlet 2.1 a introduit l'interface `javax.servlet.RequestDispatcher`. Une servlet obtient une instance de `RequestDispatcher` par le biais de la méthode `getRequestDispatcher()` sur l'objet de requête. Cette méthode retourne un `RequestDispatcher` qui peut distribuer vers le composant (servlet, JSP, fichier statique, etc.) trouvé dans le chemin d'URI indiqué :

```
public RequestDispatcher ServletRequest.getRequestDispatcher(String path)
```

Le chemin fourni peut être relatif, mais il ne peut sortir du contexte de servlet courant. Vous pouvez utiliser la méthode `getContext()` comme présenté à la section précédente pour distribuer en dehors du contexte courant. Il n'existe aucun moyen de distribuer sur un contexte d'un autre serveur. Si le chemin commence par un /, il est interprété relativement à la racine du contexte. Si le chemin contient une chaîne d'interrogation, les paramètres sont ajoutés au début de l'ensemble des paramètres du composant cible. La méthode retourne `null` si le conteneur de servlets ne peut renvoyer, pour une raison ou pour une autre, un `RequestDispatcher`.

Curieusement, il existe une méthode de même nom dans la classe `ServletContext` :

```
public RequestDispatcher ServletContext.getRequestDispatcher(String path)
```

La différence est que la version de `ServletContext` (introduite dans l'API 2.1) accepte uniquement des URL absolues (commençant par une barre oblique) alors que la ver-

sion de ServletRequest (introduite dans l'API 2.2) accepte les URL absolues et les URL relatives. Par conséquent, il n'y a aucune raison d'utiliser la méthode de ServletContext. Elle n'existe que pour des raisons historiques et on peut la considérer comme obsolète même si elle ne l'est pas officiellement.

La méthode getNamedDispatcher() de ServletContext permet d'obtenir un RequestDispatcher pour une ressource spécifiée par un nom au lieu d'un chemin :

```
public RequestDispatcher ServletContext.getNamedDispatcher(String name)
```

Cela permet de distribuer vers des ressources qui ne sont pas nécessairement disponibles publiquement. Il est possible de donner des noms aux servlets (et aussi aux pages JSP) par le biais du descripteur de déploiement d'une application web, comme expliqué au chapitre 3, *Cycle de vie d'une servlet*. La méthode retourne null si le contexte ne peut renvoyer un distributeur.

RequestDispatcher a deux méthodes, forward() et include(). La méthode forward() passe la requête complète au délégué. La méthode include() ajoute la sortie du délégué à la réponse de la servlet appelante mais lui laisse le contrôle.

Distribuer un renvoi

La méthode forward() renvoie une requête d'une servlet à une autre ressource sur le serveur. Elle permet à une servlet d'effectuer un traitement préliminaire sur une requête et de laisser une autre ressource générer la réponse. Contrairement à sendRedirect(), forward() opère au sein du serveur et le client ne peut dire qu'un renvoi s'est produit. Les informations peuvent être passées au délégué en attachant une chaîne d'interrogation ou en utilisant les attributs de requête *via* la méthode setAttribute(). L'*exemple 11-3* montre une servlet qui effectue une recherche, puis renvoie les résultats vers une autre page qui les affiche.

Exemple 11-3. Un service de traitement pour un moteur de recherche

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SearchLogic extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                          throws ServletException, IOException {
    // Nous ne définissons par le type de contenu,
    // ni n'obtenons un Writer.

    // Obtenir la chaîne à rechercher.
    String search = req.getParameter("search");

    // Trouver les URL contenant la chaîne.
    String[] results = getResults(search);

    // Mettre les résultats dans un attribut de requête.
    req.setAttribute("results", results);
```

Exemple 11-3. Un service de traitement pour un moteur de recherche (suite)

```
    // Renvoyer vers une page d'affichage.
    String display = "/servlet/SearchView";
    RequestDispatcher dispatcher = req.getRequestDispatcher(display);
    dispatcher.forward(req, res);
  }

  // Dans une utilisation réelle, cette méthode appellerait un moteur de
  // recherche et retournerait plus d'informations sur chaque résultat.
  String[] getResults(String search) {
    return new String[] { "http://www.abc.com",
                          "http://www.xyz.com" };
  }
}
```

Cette servlet joue le rôle du cerveau d'un moteur de recherche en ligne. Elle effectue la recherche du texte indiqué par le paramètre `search`, puis elle stocke les URL résultantes dans l'attribut de requête `results`. Ensuite, la servlet renvoie la requête vers un composant d'affichage. Dans notre exemple, il s'agit de */servlet/SearchView*, mais au lieu d'être figé, le chemin pourrait changer en fonction des préférences de l'utilisateur concernant la langue, les couleurs du site, un affichage pour débutant ou expert et ainsi de suite.

Une servlet de renvoi doit suivre des règles assez strictes :

- Elle peut définir des en-têtes et le code d'état, mais elle ne peut envoyer de corps de réponse au client (c'est le travail d'une inclusion). Par conséquent, `forward()` doit être appelé avant que la réponse soit validée.
- Si la réponse a déjà été validée, l'appel à `forward()` lance une `IllegalStateException`.
- Si la réponse n'a pas été validée mais que du contenu se trouve dans le tampon de réponse, celui-ci est automatiquement effacé au moment du renvoi.
- De plus, vous pouvez être créatif en substituant de nouveaux objets de requête et de réponse. La méthode `forward()` doit être appelée avec les mêmes objets de requête et de réponse que ceux passés à la méthode de service de la servlet appelante, et elle doit être appelée dans le même thread de traitement.

Le composant cible peut être écrit comme tout autre composant, comme le montre l'*exemple 11-4*.

Exemple 11-4. Frontal pour un moteur de recherche

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SearchView extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
```

Exemple 11-4. Frontal pour un moteur de recherche (suite)

```
    PrintWriter out = res.getWriter();

    // Prendre les résultats de la recherche dans un attribut de requête.
    String[] results = (String[]) req.getAttribute("results");

    if (results == null) {
      out.println("Aucun résultats.");
      out.println("Avez-vous accéder directement à cette servlet ?");
    }
    else {
      out.println("Résultats :");
      for (int i = 0; i < results.length; i++) {
        out.println(results[i]);
      }
    }

    out.println();
    out.println("URI de requête : " + req.getRequestURI());
    out.println("Chemin de contexte : " + req.getContextPath());
    out.println("Chemin de servlet : " + req.getServletPath());
    out.println("Informations de chemin : " + req.getPathInfo());
    out.println("Chaîne d'interrogation : " + req.getQueryString());
  }
}
```

Si vous essayez vous-même cette servlet, notez que l'information de chemin affichée par la servlet a été ajustée. Elle ne correspond pas à l'information de la requête originale ; elle correspond au chemin utilisé pour obtenir le distributeur :

```
URI de requête : /servlet/SearchView
Chemin de contexte :
Chemin de servlet : /servlet/SearchView
Informations de chemin : null
Chaîne d'interrogation : null
```

Cet ajustement est dû au fait que si vous renvoyez vers une servlet, cette servlet doit obtenir la même information de chemin que si elle avait été invoquée directement ; à partir de là, elle a un contrôle complet sur la requête. C'est aussi pour cela qu'il est important que le tampon de réponse soit vidé avant qu'elle soit invoquée et que la réponse ne soit pas encore validée.

Distribution par nom

Avec la distribution par chemin, si le composant cible est disponible aux composants serveur par un certain chemin, il l'est aussi aux clients avec le même chemin. Pour plus de sûreté, vous pouvez envisager de ne pas rendre un composant disponible publiquement (par exemple, en désactivant la logique d'invocation de */servlet*) et de distribuer les requêtes par le biais d'un nom à la place d'un chemin. C'est possible grâce à la méthode `getNamedDispatcher()`. L'*exemple 11-3* pourrait être réécrit avec la logique de distribution suivante :

```
// Renvoyer vers une page d'affichage.
String display = "searchView";
RequestDispatcher dispatcher = req.getNamedDispatcher(display);
dispatcher.forward(req, res);
```

Avec `getNamedDispatcher()` il faut faire attention au fait que, puisqu'aucun chemin d'URI n'est utilisé, il ne peut y avoir ni chaîne d'interrogation ni ajustement de chemin.

Renvoi contre redirection

Est-il préférable d'utiliser `forward()` ou `sendRedirect()` ? Les deux méthodes ont leurs utilisations. `forward()` fonctionne mieux quand un composant doit appliquer une logique métier et partager les résultats avec un autre composant. `sendRedirect()` fonctionne mieux quand le client doit être redirigé d'une page vers une autre. Il est tentant d'employer `forward()` au lieu de `sendRedirect()` pour des redirections simples car `forward()` opère au sein du serveur et s'exécute plus vite que `sendRedirect()` qui demande un échange avec le client. Malheureusement, cela pose des problèmes avec la gestion des URL relatives, comme le montre l'*exemple 11-5* dans lequel une servlet renvoie une requête vers la page d'accueil d'un site.

Exemple 11-5. Allez directement à la page d'accueil

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HomePageForward extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException {
    RequestDispatcher dispatcher = req.getRequestDispatcher("/index.html");
    dispatcher.forward(req, res);
  }
}
```

Si vous utilisez Tomcat pour accéder à cette servlet à l'URL *http://localhost:8080/servlet/HomePageForward*, vous noterez qu'aucune image de la page n'est affichée. La raison vient du fait que la page contient des balises `<IMG>` avec des chemins relatifs vers les fichiers d'image. Alors qu'un `sendRedirect()` aurait indiqué au client que le fichier a été servi à partir de la racine des documents, `forward()` n'a pas donné cette information au client et toutes les URL relatives sont donc invalides. La méthode `sendRedirect()` facilite aussi la distribution vers des ressources présentes dans d'autres contextes car aucune recherche `getContext()` n'est nécessaire. Notre conseil est d'employer `sendRedirect()` autant que possible et `forward()` uniquement lorsque c'est nécessaire.

Distribuer une inclusion

La méthode `include()` de `RequestDispatcher` inclut le contenu d'une ressource dans la réponse courante. Elle crée ce que l'on peut considérer comme une inclusion côté serveur par programmation. C'est différent d'un `forward()` car la servlet appelante garde

le contrôle de la réponse et elle peut inclure du contenu avant et après le contenu inclus. L'*exemple 11-6* montre une servlet qui inclut un élément de catalogue et l'affiche dans la réponse courante.

Exemple 11-6. Inclure un élément de catalogue

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NileBooks extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HTML><HEAD><TITLE>Bienvenue sur Nile</TITLE></HEAD>");
    out.println("<BODY>");

    // Afficher un élément d'un catalogue en ligne.
    out.println("Appréciez cette beauté :");

    RequestDispatcher dispatcher =
      req.getRequestDispatcher("/servlet/NileItem?item=0596000405");
    dispatcher.include(req, res);

    out.println("Et, puisque je vous aime bien, voici 20% de remise !");

    out.println("</BODY></HTML>");
  }
}
```

Comme avec forward(), les informations peuvent être passées à la ressource appelée à l'aide d'une chaîne d'interrogation attachée ou par des attributs de requête grâce à la méthode setAttribute(). L'utilisation des attributs à la place des paramètres donne la possibilité de passer des objets au lieu de simples chaînes (voir l'*exemple 11-7*).

Exemple 11-7. Inclure plusieurs éléments d'un catalogue

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NileBooks extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
```

Exemple 11-7. Inclure plusieurs éléments d'un catalogue (suite)

```
    out.println("<HTML><HEAD><TITLE>Bienvenue sur Nile</TITLE></HEAD>");
    out.println("<BODY>");

    // Afficher les éléments d'un catalogue en ligne.
    RequestDispatcher dispatcher =
      req.getRequestDispatcher("/servlet/NileItem");

    out.println("Appréciez cette beauté :");
    req.setAttribute("item", Book.getBook("0596000405"));
    dispatcher.include(req, res);

    // Supprimer l'attribut "item" après son utilisation.
    req.removeAttribute("item");

    out.println("Ou bien celle-ci :");
    req.setAttribute("item", Book.getBook("0395282659"));
    dispatcher.include(req, res);

    out.println("Et, puisque je vous aime bien, voici 20% de remise !");

    out.println("</BODY></HTML>");
  }
}

// Une classe Book simple.
class Book {
  String isbn;
  String title;
  String author;

  private static Book JSERVLET =
    new Book("0596000405", "Servlet Java, Guide du programmeur", "Hunter");

  private static Book HOBBIT =
    new Book("0395282659", "Bilbo le Hobbit", "Tolkien");

  // Simuler une recherche dans une base de données.
  public static Book getBook(String isbn) {
    if (JSERVLET.getISBN().equals(isbn)) {
      return JSERVLET;
    }
    else if (HOBBIT.getISBN().equals(isbn)) {
      return HOBBIT;
    }
    else {
      return null;
    }
  }
```

Exemple 11-7. Inclure plusieurs éléments d'un catalogue (suite)

```
  private Book(String isbn, String title, String author) {
    this.isbn = isbn;
    this.title = title;
    this.author = author;
  }

  public String getISBN() {
    return isbn;
  }

  public String getTitle() {
    return title;
  }

  public String getAuthor() {
    return author;
  }
}
```

Dans cet exemple, au lieu de passer juste un ISBN, la servlet passe un objet Book complet. Dans une utilisation réelle, les informations sur le livre viendraient d'une recherche sur l'ISBN dans une base de données. Ici, nous avons simplement figé dans le code une paire de livres.

La servlet NileItem peut recevoir l'attribut de requête item en appelant req.getAttribute("item"), comme montré à l'*exemple 11-8*.

Exemple 11-8. Afficher un élément de catalogue

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NileItem extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    // Pas de définition du type de contenu.

    PrintWriter out = res.getWriter();

    Book book = (Book) req.getAttribute("item");

    out.println("<BR>");
    if (book != null) {
      out.println("<I>" + book.getTitle() + "</I>");
      out.println(" de " + book.getAuthor());
    }
```

Exemple 11-8. Afficher un élément de catalogue (suite)

```
    else {
      out.println("<I>Aucun livre trouvé</I>");
    }
    out.println("<BR>");
  }
}
```

Une servlet incluse peut voir son contenu placé n'importe où sur la page, et, à cause de cela, elle ne peut modifier le code d'état ou les en-têtes HTTP envoyés dans la réponse. Toute tentative de modification est ignorée.

Contrairement à `forward()`, les éléments de chemin et les paramètres de la requête restent inchangés. Si le composant inclus a besoin d'accéder à ses propres éléments de chemin et paramètres, il peut les retrouver en utilisant les attributs de requête suivants définis par le serveur :

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

Les paramètres de requête et de réponse doivent être les mêmes objets que ceux passés à la méthode de service de la servlet appelante, et `include()` doit être appelée dans le même thread de traitement.

La réponse incluse doit utiliser un mécanisme de sortie qui correspond à celui de l'appelant. Si celui-ci utilise un `PrintWriter`, la ressource incluse doit utiliser un `PrintWriter`. Si l'appelant utilise un `OutputStream`, alors la ressource incluse doit utiliser un `OutputStream`. Si les mécanismes ne correspondent pas, la servlet incluse lance une `IllegalStateException`. Si vous le pouvez, assurez-vous d'employer un `PrintWriter` pour toutes les sorties de type texte et vous n'aurez aucun problème.

12

Servlets d'entreprise et J2EE

Ce chapitre traite des servlets d'entreprise. Le terme *entreprise* est très souvent employé avec Java aujourd'hui, mais que veut-il dire ? D'après ma copie du *The American Heritage Dictionary* (si vieille que son prix indiqué est de $1,95), le mot *entreprise* a trois définitions :

1. Un projet, dans un domaine et avec des risques.
2. Une société.
3. Une initiative.

De façon surprenante, c'est une définition proche de ce que les gens entendent par *Java d'entreprise* et *servlets d'entreprise*. Nous pouvons regrouper les définition traditionnelles pour en créer une plus moderne :

1. Être prêt à supporter un projet métier de grande envergure.

Autrement dit, des *servlets d'entreprise* sont des servlets conçues pour supporter des sites web orientés métier de grande taille ; des sites à fort trafic et à haute disponibilité qui ont des contraintes supplémentaires concernant l'évolutivité, l'équilibrage de charge, le support des pannes et l'intégration avec les autres technologies J2EE (*Java 2, Enterprise Edition*).

Au fur et à mesure que les servlets sont devenues plus répandues et plus robustes, et que les conteneurs de servlets sont devenus plus fiables et plus fonctionnels, un nombre croissant de sites d'entreprise se sont construits avec les servlets. L'écriture de servlets pour ces sites diffère de l'écriture de servlets pour des sites traditionnels. Dans ce chapitre, nous présenterons les contraintes et les capacités spécifiques de ces servlets d'entreprise.

Distribuer la charge

Pour les sites à fort trafic et/ou à haute disponibilité, il est souvent souhaitable de distribuer le contenu et les traitements sur plusieurs serveurs. Cette distribution permet de répartir la charge sur plusieurs serveurs, augmentant le nombre de requêtes qui peu-

vent être traitées simultanément et apportant une possibilité de basculement afin que le site ne s'arrête pas même si un composant particulier dysfonctionne.

La distribution n'est pas adaptée à tous les sites. Créer et maintenir un site réparti peut être bien plus complexe que pour un site autonome et peut aussi se révéler bien plus coûteux en termes de matériel d'équilibrage de charge et/ou de contraintes logicielles. La distribution n'améliore pas non plus vraiment les performances si le serveur n'est pas très chargé. Face à un problème de performances, il est souvent plus facile de changer de matériel et de passer à un système plus puissant que d'essayer de répartir la charge entre deux machines moins performantes.

Cependant, beaucoup de sites ont besoin de capacités qui dépassent celle d'une seule machine et qui doivent évoluer vers un niveau de disponibilité qu'une seule machine ne peut offrir. Ces sites doivent être distribués.

Comment devenir distribuable

Les contraintes de programmation sur une servlet distribuable sont bien plus strictes que celles sur une servlet non distribuable. Une servlet distribuable doit être écrite conformément à certaines règles afin que différentes instances de la servlet puissent s'exécuter sur plusieurs machines serveur. Si le programmeur suppose qu'il n'y a qu'une seule instance de la servlet, qu'un seul contexte de servlet, qu'une seule JVM ou qu'un seul système de fichiers, de graves problèmes peuvent survenir.

Pour apprendre à distribuer des servlets, examinons la technologie des EJB (*Enterprise JavaBeans*), un modèle de composant côté serveur pour l'implémentation d'objets métier distribués et la technologie au cœur de J2EE. Les EJB sont conçus dès le départ comme des objets distribués. Un EJB implémente une logique métier et laisse le *conteneur* (globalement le serveur) dans lequel il s'exécute prendre en charge les services comme les transactions, la persistance, la concurrence et la sécurité. Un EJB peut être distribué sur plusieurs machines serveur et il peut passer d'une machine à l'autre selon la volonté du conteneur. Pour permettre ce modèle de distribution, les EJB doivent suivre un ensemble de règles strictes définies par la spécification et dictant ce qu'ils peuvent et ne peuvent pas faire.[1]

Les servlets ne disposent pas de cet ensemble de règles. Cela provient de leur emploi comme composants côté serveur frontaux pour communiquer avec le client et appelant les EJB distribués, mais n'étant pas elles-mêmes distribuées. Cependant, pour les sites à fort trafic ou pour les sites ayant besoin de haute disponibilité, les servlets doivent elles aussi être distribuées. Nous espérons que les futures versions de l'API Servlet incluent une définition plus précise de l'implémentation des conteneurs de servlets distribuées.

Voici nos règles générales pour l'écriture de servlets à déployer dans un environnement distribué :

- Considérez que les différentes instances de la servlet peuvent chacune exister dans des JVM et/ou des machines différentes. Ainsi, les variables d'instance et les variables statiques ne doivent pas servir à stocker un état. Tout état doit être placé dans

1. Pour plus d'informations sur les Enterprise JavaBeans, allez voir *http://java.sun.com/products/ejb* et le livre *Enterprise JavaBeans* de Richard Monson-Haefel (Éditions O'Reilly).

une ressource externe comme une base de données ou un EJB (le nom de la servlet peut être utilisé pour la recherche).

- Considérez que différentes instances de `ServletContext` peuvent exister sur chaque JVM et/ou machine différente. Aussi, le contexte ne doit pas servir à stocker l'état de l'application. Tout état doit être placé dans une ressource externe comme une base de données ou un EJB (le chemin du contexte de servlet peut être utilisé pour la recherche).
- Considérez que tout objet placé dans un `HttpSession` peut être déplacé vers une (ou accédé à partir d'une) machine différente. Par exemple, l'objet peut implémenter `java.io.Serializable`. Sachez que les sessions pouvant migrer, l'événement de fin de liaison à une session peut survenir sur une machine différente de celle sur laquelle l'événement de liaison est survenu !
- Considérez que les fichiers peuvent ne pas exister sur toutes les machines serveur. Ainsi, vous devez éviter d'utiliser le paquetage `java.io` pour les accès aux fichiers et employer à la place le mécanisme `getServletContext().getResource()`, ou vous assurer que tous les fichiers sont répliqués sur toutes les machines serveur.
- Considérez que la synchronisation n'est pas globale et ne fonctionne que sur la JVM locale.

Une application web dont les composants suivent ces règles peut être marquée comme *distribuable*. Ce marquage permet au serveur de déployer l'application sur de multiples machines serveur. Il se trouve dans le descripteur de déploiement *web.xml* sous forme d'une balise `<distributable/>` vide située entre la description de l'application et ses paramètres de contexte :

```
<web-app>
  <description>
    Toutes les servlets et les pages JSP sont prêtes pour un
    déploiement distribué
  </description>

  <distributable/>

  <context-param>
    <!-- ... -->
  </context-param>
</web-app>
```

Par défaut, les applications ne sont pas distribuables, afin que les programmeurs de servlets occasionnels puissent les développer sans s'occuper des règles supplémentaires du déploiement distribué. Marquer une application comme distribuable ne signifie pas nécessairement que l'application sera répartie sur différentes machines. Cela indique uniquement sa capacité à être répartie. Voyez-le comme une certification fournie par le programmeur.

Les serveurs n'obligent pas une application distribuée à respecter la plupart des règles précédentes. Par exemple, rien n'empêche une servlet d'utiliser les variables d'instance et les variables statiques, ni de stocker des objets dans son `ServletContext`, et une servlet peut toujours accéder à des fichiers à l'aide du paquetage `java.io` package. C'est au programmeur de s'assurer que ces possibilités sont correctement employées. La seule

obligation que le serveur peut faire respecter est le lancement d'une `IllegalArgumentException` si un objet lié au `HttpSession` n'implémente pas `java.io.Serializable` (cette obligation est même optionnelle car, comme nous le verrons plus loin, un serveur compatible J2EE doit autoriser le stockage d'autres types d'objets dans la session).

Plusieurs styles de distribution

La distribution de servlets (souvent appelée le *clustering*) est une fonctionnalité optionnelle d'un conteneur de servlets. Un conteneur de servlets qui supporte le clustering est libre de l'implémenter comme il l'entend. Il existe quatre architectures standard, listées de la plus simple à la plus complexe :

1. Pas de clustering. Toutes les servlets s'exécutent dans une seule JVM et le marqueur `<distributable/>` est essentiellement ignoré. Cette conception est simple et fonctionne parfaitement pour un site standard. Le serveur Tomcat autonome fonctionne de cette manière.
2. Support du clustering, pas de migration de session et pas de basculement de session. Les servlets d'une application web marquées `<distributable/>` peuvent s'exécuter sur plusieurs machines. Les requêtes non-session sont distribuées aléatoirement (peut-être d'après un certain poids). Les requêtes session sont « collantes » et liées à un serveur particulier sur lequel elles démarrent. Les données de session ne passent pas d'une machine à une autre. L'avantage est que les sessions peuvent détenir des données non transférables (non `Serializable`), l'inconvénient est que les sessions ne peuvent migrer sur des serveurs sous-utilisés et un dysfonctionnement de serveur peut conduire à des sessions invalides. C'est l'architecture employée par Apache/JServ et Apache/Tomcat. Les sessions sont liées à une machine particulière au travers d'un mécanisme dans lequel le connecteur `mod_jserv/mod_jk` d'Apache utilise une partie de l'identifiant de session pour indiquer quel serveur JServ ou Tomcat détient la session. Il est aussi possible d'utiliser de multiples instances d'Apache avec le support d'un matériel ou d'un logiciel de répartition de charge.
3. Support du clustering, avec migration de session et pas de basculement de session. Cette architecture fonctionne de la même façon que la précédente excepté qu'une session peut migrer d'un serveur à un autre afin d'améliorer la répartition de la charge. Pour éviter les problèmes de concurrence, toute migration de session se passe entre des requêtes utilisateur. La spécification des servlets stipule qu'« Au sein d'une application marquée comme distribuable, toutes les requêtes faisant partie d'une session ne peuvent être traitées que dans une seule machine virtuelle à un moment donné. » Tous les objets placés dans une session qui peut migrer doivent implémenter `java.io.Serializable` ou doivent être transférables d'une manière ou d'une autre.

- Support du clustering, avec migration de session et avec basculement de session. Un serveur implémentant cette architecture a la possibilité de dupliquer le contenu d'une session afin qu'un dysfonctionnement d'un composant individuel ne perturbe pas obligatoirement la session d'un utilisateur. Les problèmes de cette architecture concerne la coordination efficace et le flux d'informations. La plupart des serveurs haut de gamme sont conformes à cette architecture.

Les détails d'implémentation du clustering varient suivant les serveurs et constituent un point de concurrence pour les fournisseurs. Reportez-vous à la documentation de votre serveur pour savoir quel niveau de clustering il supporte. Une autre fonctionnalité utile à examiner est la *persistance de session*, la sauvegarde en arrière-plan des informations de session sur un disque ou une base de données, qui permet aux informations de survivre aux redémarrages et aux dysfonctionnements du serveur.

Intégration avec J2EE

Jusqu'à présent, les servlets ont été utilisées comme une technologie construite au-dessus d'une base Java standard. Cependant, les servlets ont une autre vie, dans laquelle elles agissent comme partie intégrante de ce que l'on appelle *Java 2, Enterprise Edition*, ou *J2EE*.[2] J2EE 1.2 rassemble plusieurs API côté serveur dont l'API Servlet 2.2, les JSP 1.1, les EJB, JavaMail, JMS (*Java Messaging Service*), JTA (*Java Transactions*), CORBA, JDBC, JAXP (*Java API for XML Parsing*) et JNDI (*Java Naming and Directory Interface*). J2EE fait bien plus que de réunir tous ces composants car il définit comment ces technologies peuvent coopérer et s'utiliser l'une l'autre. De plus, J2EE apporte la certification que certains serveurs d'application sont compatibles avec J2EE, c'est-à-dire qu'ils apportent tous les services requis ainsi que leur interconnexion.

Répartition des tâches dans J2EE

J2EE découpe le développement d'applications d'entreprise en six rôles distincts. Bien entendu, un individu peut participer à plusieurs rôles et plusieurs individus peuvent travailler sur un même rôle.

Fournisseur de produits J2EE

Le fournisseur de systèmes d'exploitation, le fournisseur du système de gestion de bases de données, le fournisseur de serveurs d'applications et/ou le fournisseur de serveurs web. Le fournisseur de produits apporte une implémentation des API J2EE et des outils pour le déploiement et la gestion des applications.

Fournisseur de composants d'application

L'auteur des servlets, des EJB et des autres codes de l'application ainsi que du contenu général comme le HTML. (Autrement dit, vous.)

L'assembleur d'application

Il prend les composants de l'application et (à l'aide des outils du fournisseur de produits) les place sous une forme adaptée à leur déploiement. Une partie de cet assemblage décrit les dépendances externes de l'application qui peuvent changer d'un déploiement à un autre, comme la base de données ou les informations d'ouverture de session.

Le déployeur

Il prend le résultat du travail de l'assembleur et (à l'aide des outils du fournisseur de

2. La plupart des gens prononcent J2EE comme J-2-E-E, mais ceux qui le connaissent bien chez Sun disent simplement « j-2-e ».

produits) installe, configure et exécute l'application. La tâche de configuration nécessite de résoudre les dépendances externes signalées par l'assembleur.

L'administrateur système

Il configure et administre l'infrastructure réseau afin que les applications fonctionnent.

Le fournisseur d'outils

Il crée des outils pour supporter le développement J2EE, plus avancés que ceux du fournisseur de produits.

Le découpage des tâches entre le fournisseur de composants, l'assembleur et le déployeur a un impact important sur la façon dont nous (en tant que programmeurs de servlets dans le rôle du fournisseur de contenu) nous comportons. En particulier, nous devons concevoir notre code afin que les dépendances externes soient claires pour l'assembleur et nous devons de plus utiliser les mécanismes permettant au déployeur de résoudre ces dépendances sans modifier les fichiers reçus de l'assembleur. Cela signifie que le déployeur ne doit pas modifier le fichier *web.xml* ! Pourquoi ? Parce que les applications J2EE sont assemblées dans des fichiers *.ear* (*Enterprise Archive*) dans lesquels on trouve le fichier *web.xml* de l'application web mais dans une partie non modifiable.

Cela semble plus complexe que ça ne l'est en réalité. J2EE fournit un mécanisme standard pour apporter cette abstraction en utilisant JNDI et quelques balises spéciales dans le descripteur de déploiement *web.xml*. JNDI est un mécanisme de recherche d'objets, une façon de lier des objets à certains chemins et de les retrouver ensuite par ces chemins. Vous pouvez le voir comme un registre RMI, excepté qu'il est plus général avec un support d'un plus grand nombre de services, y compris LDAP et NIS (et même, en fait, le registre RMI !). Un assembleur déclare les dépendances externes dans le fichier *web.xml* en utilisant des balises spéciales, un déployeur résout ces dépendances en utilisant des outils spécifiques au serveur, et, à l'exécution, notre code Java utilise l'API JNDI pour accéder aux ressources externes, gentiment placées là par le serveur J2EE. Tous les objectifs sont remplis : notre code Java reste portable sur les serveurs compatibles J2EE et le déployeur peut résoudre les dépendances externes du code sans modifier les fichiers reçus de l'assembleur. Il reste même assez de liberté pour que les fournisseurs de serveurs entrent en concurrence sur l'implémentation des standards.

Entrées de l'environnement

Les paramètres d'initialisation du contexte remplissent un rôle utile avec les servlets, mais ils ont un problème dans le modèle J2EE : toute modification de la valeur d'un paramètre nécessite une modification du fichier *web.xml*. Pour les paramètres qui doivent changer au moment du déploiement, il est préférable d'utiliser les entrées de l'environnement, indiquées par la balise `<env-entry>`. La balise `<env-entry>` peut contenir les éléments `<description>`, `<env-entry-name>`, `<env-entry-value>` et `<env-entry-type>`. L'exemple suivant indique si l'application doit autoriser l'envoi des codes PIN par courrier :

```
<env-entry>
  <description>Envoyer les codes PIN par courrier</description>
  <env-entry-name>mailPincode</env-entry-name>
  <env-entry-value>false</env-entry-value>
```

```
    <env-entry-type>java.lang.Boolean</env-entry-type> <!-- NCC -->
  </env-entry>
```

<description> explique au déployeur le but de cette entrée. Il est optionnel mais il vaut mieux le mettre. <env-entry-name> est utilisé par le code Java dans la recherche JNDI. <env-entry-value> définit la valeur par défaut à présenter au déployeur. Il est optionnel, mais s'il n'est pas donné, c'est au déployeur de donner une valeur par défaut. <env-entry-type> représente le nom de classe complet (NCC) de l'entrée. Le type peut être un String, un Byte, un Short, un Integer, un Long, un Boolean, un Double ou un Float (tous avec leur qualification java.lang complète). Le type aide le déployeur à déterminer ce qui est attendu. Si vous connaissez le descripteur de déploiement des EJB, ces balises doivent vous sembler familières ; elles ont le même nom et la même sémantique dans les EJB.

Voici le code Java pour retrouver les valeurs <env-entry> avec JNDI :

```
Context initCtx = new InitialContext();
Boolean mailPincode = (Boolean) initCtx.lookup("java:comp/env/mailPincode");
```

Toutes les entrées sont placées par le serveur dans le contexte java:comp/env. Si vous débutez avec JNDI, vous pouvez voir cela comme une base d'URL ou un répertoire du système de fichiers. Le contexte java:comp/env est en lecture seule et unique à chaque application web. Si deux applications web différentes définissent la même entrée de l'environnement, ces entrées n'entrent donc pas en collision. Les abréviations dans le contexte signifient *component environment*.

L'*exemple 12-1* montre une servlet qui affiche toutes ses entrées de l'environnement en utilisant l'API JNDI pour parcourir le contexte java:comp/env.

Exemple 12-1. Analyser le contexte java:comp/env

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import javax.naming.*;

public class EnvEntrySnoop extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    try {
      Context initCtx = new InitialContext();
      NamingEnumeration enum = initCtx.listBindings("java:comp/env");

      // Nous utilisons les méthodes du JDK 1.2 ; c'est parfait car J2EE
      // nécessite le JDK 1.2.
      while (enum.hasMore()) {
        Binding binding = (Binding) enum.next();
```

Exemple 12-1. Analyser le contexte java:comp/env (suite)

```
        out.println("Nom : " + binding.getName());
        out.println("Type: " + binding.getClassName());
        out.println("Valeur : " + binding.getObject());
        out.println();
      }
    }
    catch (NamingException e) {
      e.printStackTrace(out);
    }
  }
}
```

En prenant l'entrée *web.xml* précédente, la servlet doit générer :

```
Nom : mailPincode
Type : java.lang.Boolean
Valeur : false
```

Rappelez-vous qu'un serveur qui ne supporte pas J2EE n'est pas obligé de supporter ces balises ou les autres balises dont nous avons parlées dans cette section.

Références aux composants EJB

Quand l'objet de l'entrée de l'environnement est un composant EJB, il faut utiliser une balise `<ejb-ref>` spéciale. Elle apporte aux servlets un moyen d'obtenir une référence à un EJB à l'aide d'un nom abstrait. Le déployeur s'assure de la disponibilité du bean adéquat à l'exécution d'après les contraintes données par la balise `<ejb-ref>`. Cette balise peut contenir les éléments `<description>`, `<ejb-ref-name>`, `<ejb-ref-type>`, `<home>`, `<remote>` et `<ejb-link>`. Voici une balise `<ejb-ref>` type :

```
<ejb-ref>
  <description>Cabine d'un paquebot</description>
  <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.titan.cabin.CabinHome</home>
  <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

Ces balises ont leurs équivalents dans les EJB et, en fait, cet exemple a été emprunté au livre *Enterprise JavaBeans* de Richard Monson-Haefel (Éditions O'Reilly). `<description>` aide le déployeur, il est optionnel mais conseillé. `<ejb-ref-name>` donne le nom de recherche JNDI. Il est recommandé (mais pas obligatoire) que le nom soit placé dans le sous-contexte `ejb/`, donnant au bean le chemin complet `java:comp/env/ejb/CabinHome`. `<ejb-ref-type>` doit prendre la valeur `Entity` ou `Session`, les deux types d'EJB.[3] Enfin, l'élément `<home>` spécifie le nom de classe complet de l'interface du Home de

3. La spécification de l'API Servlet 2.2 stipule que « l'élément `ejb-ref-type` contient le type de classe Java attendue pour l'EJB référencé. » C'est une erreur confirmée. L'objectif réel est celui indiqué ici.

l'EJB, alors que l'élément <remote> spécifie le nom de classe complet de l'interface distante de l'EJB.

Une servlet obtient une référence au bean Cabin à l'aide du code suivant :

```
InitialContext initCtx = new InitialContext();
Object ref = initCtx.lookup("java:comp/env/ejb/CabinHome");
CabinHome home =
  (CabinHome) PortableRemoteObject.narrow(ref, CabinHome.class);
```

Si l'assembleur écrivant le fichier *web.xml* a à l'esprit un composant EJB spécifique pour une référence d'EJB, cette information peut être donnée au déployeur par l'ajout de l'élément <ejb-link> optionnel. L'élément <ejb-link> doit faire référence au <ejb-name> d'un composant EJB enregistré dans le descripteur de déploiement de la même application J2EE. Le déployeur a le choix d'utiliser la suggestion ou de l'écraser. Voici une entrée *web.xml* revue :

```
<ejb-ref>
  <description>Cabine d'un paquebot</description>
  <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.titan.cabin.CabinHome</home>
  <remote>com.titan.cabin.Cabin</remote>
  <ejb-link>CabinBean</ejb-link>
</ejb-ref>
```

Références à des fabriques de ressources externes

Enfin, quand l'entrée de l'environnement est une fabrique de ressources, il faut utiliser la balise <resource-ref>. Une fabrique est un objet qui crée d'autres objets à la demande. Une fabrique de ressources crée des objets représentant des ressources, comme des connexions à une base de données ou des files de messages.

La balise <resource-ref> peut contenir les éléments <description>, <res-ref-name>, <res-type> et <res-auth>. Voici une balise <resource-ref> type :

```
<resource-ref>
  <description>Base de données principale</description>
  <res-ref-name>jdbc/primaryDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>CONTAINER</res-auth>
</resource-ref>
```

<description> aide elle aussi le déployeur, elle est optionnelle mais conseillée. <res-ref-name> donne le nom de recherche JNDI. Il est recommandé mais pas obligatoire de placer des fabriques de ressources dans un sous-contexte qui décrit le type des ressources :

- jdbc/ pour une fabrique de javax.sql.DataSource JDBC ;
- jms/ pour un javax.jms.QueueConnectionFactory ou un javax.jms.TopicConnectionFactory JMS ;
- mail/ pour une fabrique de javax.mail.Session JavaMail ;
- url/ pour une fabrique de java.net.URL.

L'élément <res-type> spécifie le nom de classe complet de la fabrique de ressources (non la ressource créée). Les types de fabriques de la liste précédente sont des types standard. Un serveur a le choix de supporter d'autres types ; les fabriques utilisateur ne peuvent être utilisées. La spécification J2EE 1.3 à venir propose un mécanisme de « connecteur » qui étend ce modèle pour les fabriques définies par l'utilisateur.

<res-auth> indique au serveur qui est responsable de l'authentification. Il peut prendre deux valeurs : CONTAINER ou SERVLET. Si CONTAINER est donné, le conteneur de servlets (le serveur J2EE) prend en charge l'authentification avant de lier la fabrique à JNDI, à l'aide des certificats fournis par le déployeur. Si SERVLET est spécifié, la servlet doit prendre en charge l'authentification par programmation. Par exemple :

```
InitialContext initCtx = new InitialContext();
DataSource source =
  (DataSource) initCtx.lookup("java:comp/env/jdbc/primaryDB");

// Si "CONTAINER".
Connection con1 = source.getConnection();

// Si "SERVLET".
Connection con2 = source.getConnection("user", "password");
```

Ces balises ont leur équivalent dans le descripteur de déploiement des EJB. La seule différence est que dans les EJB, les deux valeurs possibles pour <res-auth> sont Container et Application (notez l'inexplicable différence de casse).

Distribution de servlets dans J2EE

La dernière différence entre des servlets dans un environnement autonome et des servlets dans un environnement J2EE implique une modification subtile des règles de distribution de session. Alors qu'un serveur web standard n'est obligé de supporter que les objets java.io.Serializable dans la session d'une application distribuable, un serveur compatible J2EE qui supporte un conteneur de servlets distribuées doit aussi supporter plusieurs autres types d'objets :

- tout javax.ejb.EJBObject ;
- tout javax.ejb.EJBHome ;
- tout javax.transaction.UserTransaction ;
- le javax.naming.Context pour java:comp/env.

Toutes ces interfaces n'implémentent pas Serializable. Pour transférer les objets, le conteneur peut utiliser son propre mécanisme personnalisé, peut-être basé sur la sérialisation. D'autres types de classes peuvent être supportés par le serveur, mais ceux-là sont les seuls types garantis.

13

Internationalisation

En dépit de son nom, le World Wide Web a encore une longue route à parcourir avant de vraiment s'étendre au monde entier. Bien sûr, les liens physiques amènent le contenu dans à peu près tous les pays du globe. Mais pour que l'on puisse le considérer vraiment comme une ressource mondiale, ce contenu web doit pouvoir être lu par la personne qui le reçoit, ce qui n'arrive pas souvent avec le grand nombre de pages uniquement en anglais que l'on trouve aujourd'hui.

Cependant la situation commence à changer. Beaucoup de sites web importants proposent des zones conçues pour des langues autres que l'anglais. Par exemple, la page d'accueil de Netscape est disponible aux visiteurs anglais à *http://home.netscape.com/index.html*, aux visiteurs français à *http://home.netscape.com/fr/index.html* et en une dizaine d'autres langues à une dizaine d'autres URL.

Plusieurs serveurs web supportent aussi une solution transparente, où une seule URL peut être utilisée pour voir le même contenu en différentes langues, la langue étant choisie en fonction des préférences du client. La langue utilisée dépend de la configuration de votre navigateur[1]. Bien que cette technique crée l'impression qu'une traduction dynamique a lieu, en réalité le serveur dispose tout simplement de plusieurs versions statiques de pages nommées de façon spéciale.

Alors que ces techniques fonctionnent bien pour des documents statiques, elles ne règlent pas le problème de l'internationalisation et de la localisation d'un contenu dynamique. C'est le sujet de ce chapitre. Nous examinons ici comment les servlets peuvent utiliser les possibilités d'internationalisation ajoutées au JDK 1.1 pour que votre site web devienne vraiment mondial.

Tout d'abord, nous allons parler terminologie. L'*internationalisation* (un mot que l'on réduit souvent à *I18N* car il commence par un « I », se termine par un « n » et possède 18 lettres entre les deux) consiste à rendre un programme suffisamment souple pour qu'il

1. Cependant, plusieurs navigateurs anciens n'acceptent pas la personnalisation de la langue. Par exemple, cette possibilité n'est apparue que dans Netscape Navigator 4 et Microsoft Internet Explorer 4.

fonctionne avec n'importe quelle *locale*. La *localisation* (souvent raccourcie à *L10N*) consiste à organiser un programme pour qu'il fonctionne avec une locale spécifique. Ce chapitre, dans sa majeure partie, couvre l'internationalisation des servlets. Nous n'examinerons la localisation uniquement dans le cas des dates, heures, nombres et autres objets pour lesquels il existe un support de localisation intégré à Java.

Langues de l'Europe de l'ouest

Commençons par regarder comment une servlet sort une page écrite dans une langue de l'Europe de l'ouest telle que l'anglais, l'espagnol, l'allemand, le français, l'italien, le hollandais, le norvégien, le finlandais ou le suédois. Pour notre exemple, nous allons écrire « Bonjour tout le monde ! » en espagnol, en générant une page similaire à celle montrée à la *figure 13-1*.

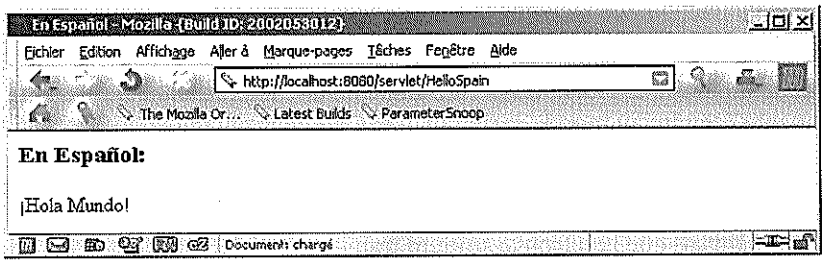


Figure 13-1. En Español: ¡Hola Mundo!

Notez l'utilisation des caractères spéciaux *ñ* et *¡*. Ce type de caractères, bien que peu répandu en anglais, sont bien plus fréquents dans les langues de l'Europe de l'ouest. Les servlets ont deux façons de générer ces caractères : avec les entités caractères de HTML ou avec les séquences d'échappement Unicode.

Entités caractères de HTML

HTML 2.0 a apporté la possibilité d'afficher sous forme d'un seul caractère dans une page HTML des séquences de caractères spécifiques. Les séquences, appelées *entités caractères*, commencent par une esperluette (&) et se terminent par un point-virgule (;). Les entités caractères peuvent être soit nommées soit numérotées. Par exemple, l'entité caractère nommée `ñ` représente *ñ* et `¡` représente *¡*. Une liste complète des caractères spéciaux et de leur nom est donnée à l'annexe E, *Entités caractères*. L'*exemple 13-1* montre une servlet qui utilise les entités nommées pour dire bonjour en espagnol.

Exemple 13-1. Bonjour aux Espagnols, en utilisant les entités caractères nommées

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloSpain extends HttpServlet {
```

Exemple 13-1. Bonjour aux Espagnols, en utilisant les entités caractères nommées (suite)

```
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "es");

    out.println("<HTML><HEAD><TITLE>En Espa&ntilde;ol</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H3>En Espa&ntilde;ol:</H3>");
    out.println("&iexcl;Hola Mundo!");
    out.println("</BODY></HTML>");
  }
}
```

Vous avez peut-être noté que, en plus d'utiliser les entités caractères, cette servlet met l'en-tête `Content-Language` à la valeur `es`. L'en-tête `Content-Language` est utilisé pour indiquer la langue du corps qui suit. Dans ce cas, la servlet utilise l'en-tête pour indiquer au client que la page est écrite en espagnol (Español). La plupart des clients ignorent cette information, mais il est poli de l'envoyer. Les langues sont toujours représentées par des abréviations de deux caractères en minuscule. Pour une liste complète, voyez la norme ISO-639 à *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*.

Les entités caractères peuvent aussi être représentées par un nombre. Par exemple, `ñ` représente *ñ* et `¡` représente *¡*. Le nombre correspond à la valeur décimale ISO-8859-1 (Latin-1) du caractère, sur lequel nous reviendrons en détail plus loin dans ce chapitre. Une liste complète des valeurs numériques pour les entités caractères est aussi donnée à l'annexe E. L'*exemple 13-2* montre `HelloSpain` réécrit pour utiliser des entités numériques.

Exemple 13-2. Bonjour aux Espagnols, en utilisant les entités caractères numérotées

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloSpain extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "es");

    out.println("<HTML><HEAD><TITLE>En Espa&#241;ol</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H3>En Espa&241;ol:</H3>");
    out.println("&#161;Hola Mundo!");
    out.println("</BODY></HTML>");
  }
}
```

Malheureusement, il y a un problème important lorsque l'on utilise les entités caractères : cela ne marche que pour les pages HTML. Si la sortie de la servlet n'est pas une page HTML, la page ressemble à ce qui est montré à la *figure 13-2*. Pour gérer une sortie non-HTML, nous avons besoin des *échappements Unicode*.

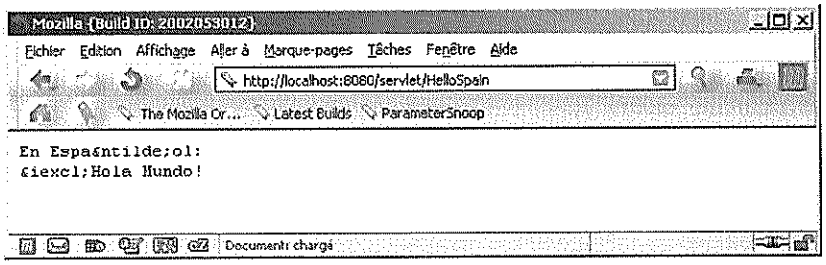


Figure 13-2. Pas vraiment de l'espagnol

Échappements Unicode

En Java, tous les caractères, chaînes de caractères et identifiants sont codés en interne par des caractères Unicode 2.0 sur 16 bits (2 octets). Unicode a été créé par l'Unicode Consortium, qui décrit la norme de la façon suivante (allez voir à *http://www.unicode.org*) :

> L'Unicode Worldwide Character Standard est un système de codage des caractères conçu pour l'échange, le traitement et l'affichage des textes écrits dans diverses langues du monde moderne. De plus, il supporte les textes classiques et historiques de plusieurs langues écrites.
>
> Dans sa version courante (2.0), la norme Unicode contient 38 885 caractères de code différent dérivés de Supported Scripts. Ces caractères couvrent les principales langues écrites des Amériques, de l'Europe, du Moyen Orient, de l'Afrique, de l'Inde, de l'Asie et du Pacifique.

Pour plus d'informations sur Unicode, allez voir à *http://www.unicode.org*. Vous pouvez aussi lire *The Unicode Standard, Version 2.0* (Addison-Wesley). Notez que même si Unicode 3.0 est disponible, Java supporte toujours la version 2.0.

L'utilisation d'Unicode dans Java est très importante pour ce chapitre car elle signifie qu'une servlet peut représenter en interne à peu près tous les caractères généralement utilisés dans le langage écrit. Nous pouvons représenter les caractères Unicode 16 bits dans du code source ASCII-US 7 bits en utilisant les échappements Unicode de la forme `\uxxxx`, où *xxxx* est une séquence de quatre chiffres hexadécimaux. Le compilateur Java interprète chaque séquence d'échappement Unicode comme un seul caractère.

Il est pratique, mais ce n'est pas une coïncidence, que les 256 premiers caractères d'Unicode (`\u0000` à `\u00ff`) correspondent aux 256 caractères de l'ISO-8859-1 (Latin-1). Ainsi, le caractère *ñ* peut s'écrire `\u00f1` et le caractère *¡* peut s'écrire `\u00a1`. Une liste complète des séquences d'échappement Unicode pour les caractères ISO-8859-1 est aussi incluse dans l'annexe E. L'*exemple 13-3* montre `HelloSpain` réécrit pour utiliser les échappements Unicode.

Exemple 13-3. Bonjour aux Espagnols, en utilisant les échappements Unicode

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloSpain extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "es");

    out.println("En Espa\u00f1ol:");
    out.println("\u00a1Hola Mundo!");
  }
}
```

La sortie de cette servlet s'affiche correctement lorsqu'elle est utilisée dans une page HTML ou dans du texte pur.

Se conformer aux habitudes locales

Maintenant nous savons comment utiliser les entités caractères HTML et les échappements Unicode pour afficher les caractères des langues de l'Europe de l'ouest. La question suivante est : que disons-nous dans ces langues ? En général, c'est un problème de traduction qu'il vaut mieux laisser à une équipe de localisation. Cependant, dans certains cas, Java peut apporter de l'aide.

Par exemple, supposons qu'en plus de dire « Bonjour tout le monde », nous devions donner l'heure courante dans un format naturel pour le destinataire. Ce qui pourrait être un problème de formatage difficile est en fait assez facile car le JDK 1.1 fournit un support intégré pour localiser des objets dynamiques comme les dates et l'heure.

L'astuce consiste à utiliser une instance de `java.text.DateFormat` adaptée au public visé. Un objet `DateFormat` peut convertir un `Date` en un `String` correctement localisée. Par exemple, une estampille écrite en anglais « February 16, 1998 12:36:18 PM PST » sera écrite en espagnol « 16 de febrero de 1998 12:36:18 GMT-08:00 ».

Un objet `DateFormat` est créé en utilisant une méthode qui accepte un style de formatage (court, moyen, long, complet) et un objet `java.util.Locale` identifiant le public cible (Anglais U.S., Chinois continental, etc.). Le constructeur de `Locale` le plus commun accepte deux paramètres : une abréviation de la langue sur deux caractères en minuscules (comme vu précédemment) et un code de pays sur deux caractères en majuscules comme défini par l'ISO-3166 (disponible à *http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*). Une chaîne vide pour le code du pays indique le pays par défaut pour la langue.

L'*exemple 13-4* montre la servlet `HelloSpain` utilisant un objet `DateFormat` pour afficher l'heure courante dans un format naturel à un public espagnol.

Exemple 13-4. Bonjour aux Espagnols, avec l'heure localisée

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloSpain extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "es");

    Locale locale = new Locale("es", "");
    DateFormat fmt = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                    DateFormat.LONG,
                                                    locale);
    fmt.setTimeZone(TimeZone.getDefault());

    out.println("En Espa\u00f1ol:");
    out.println("\u00a1Hola Mundo!");
    out.println(fmt.format(new Date()));
  }
}
```

Cette servlet commence par créer un `Locale` qui représente un environnement espagnol générique. Ensuite, elle utilise ce `Locale` pour créer une instance de `DateFormat` qui formate les dates en espagnol. Puis, elle définit le fuseau horaire à celui par défaut (celui du serveur). La raison est que, par défaut, un objet `DateFormat` formate son heure afin qu'elle corresponde au fuseau horaire dans lequel il suppose que le receveur se trouve, dans ce cas l'Espagne. Cette servlet ne pouvant en être certaine, elle remplace ce comportement par défaut et utilise le fuseau horaire du serveur. Bien sûr, il vaudrait mieux utiliser le fuseau horaire correspondant à celui du client, mais ce n'est pas actuellement possible sans que celui-ci donne plus d'informations. Enfin, après avoir écrit son « Bonjour tout le monde », la servlet affiche la date et l'heure correctement mises en forme. La sortie est montrée à la *figure 13-3*.

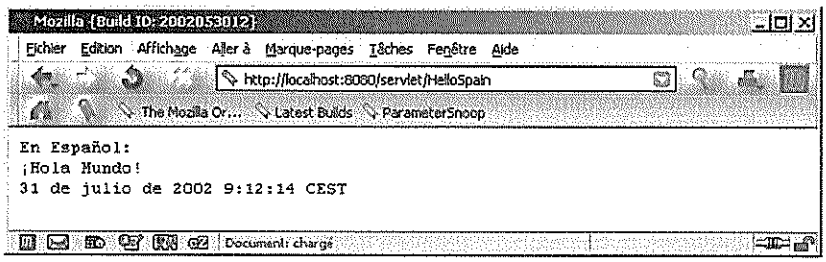


Figure 13-3. Hola Mundo con Tiempo

Cet exemple ne donne qu'un bref aperçu des possibilités de formatage dynamique de Java. Si vous êtes intéressé par des mises en forme plus complexes, il existe plusieurs autres classes dans le package `java.text` que vous pourriez trouver utiles. Regardez en particulier celles qui étendent `java.text.Format`.

Langues hors Europe de l'ouest

Continuons par examiner comment une servlet peut sortir une page dans une langue qui n'est pas de l'Europe de l'ouest, telle que le russe, le japonais, le chinois, le coréen ou l'hébreux. Pour travailler avec ces langues, nous devons commencer par comprendre ce qui se passe en coulisses dans nos exemples précédents.

Charsets

Commençons par examiner la situation du point de vue du navigateur. Imaginons que nous devions effectuer son travail. Vous faites une requête sur une certaine URL et recevez une réponse. Cette réponse n'est rien de plus qu'une longue suite d'octets. Comment savez-vous de quelle manière afficher cette réponse ?

Le plus courant, et en fait la façon de faire par défaut, est de supposer que chaque octet représente un des 256 caractères possibles et que l'on peut trouver le caractère représenté par un octet en recherchant cette valeur dans une table. La table par défaut est spécifiée par la norme ISO-8859-1, aussi appelée Latin-1. Elle contient une correspondance octet-à-caractère pour les caractères les plus usités dans les langues de l'Europe de l'ouest. Donc, par défaut, vous (agissant comme le navigateur) pouvez recevoir une suite d'octets et les convertir en une suite de caractères de l'Europe de l'ouest.

Maintenant, que faites-vous si vous voulez recevoir un texte qui n'est pas écrit dans une langue de l'Europe de l'ouest ? Vous devez prendre une longue suite d'octets de la réponse et l'interpréter différemment, en utilisant une autre correspondance entre une séquence d'octets et un caractère. D'un point de vue technique, il vous faut utiliser un *charset*[2] différent. Il existe un nombre infini de charsets potentiels. Heureusement, seule une dizaine est couramment utilisée.

Certains charsets utilisent des caractères sur un octet tout comme l'ISO-8859-1, mais avec une correspondance octet-à-caractère différente. Par exemple, l'ISO-8859-5 définit une correspondance octet-à-caractère pour l'alphabet cyrillique (russe), tandis que l'ISO-8859-8 définit une correspondance pour l'alphabet hébreux[3].

D'autres charsets utilisent des caractères multi-octets, dans lesquels un caractère peut être représenté par plus d'un octet. C'est plus courant avec les langues qui contiennent des milliers de caractères, comme le chinois, le japonais et le coréen, souvent désignées ensemble sous le nom CJK (*Chinese, Japanese, Korean*). Les charsets utilisés pour afficher ces

---

2. Un *charset* (une correspondance entre une suite d'octets et un caractère) n'est pas la même chose qu'un *jeu de caractères* (un ensemble de caractères). Allez voir la RFC 2278 à *http://www.ietf.org/rfc/rfc2278.txt* pour une explication complète.

3. Il est utile de noter que, pour à peu près tous les charsets, les valeurs entre 0 et 127 représentent les caractères ASCII-US, permettant d'ajouter un texte anglais dans une page écrite dans presque n'importe quelle langue.

langues incluent Big5 (chinois), Shift_JIS (japonais) et EUC-KR (coréen). Une table donnant la liste des langues et leurs charsets correspondant se trouve à l'annexe F, *Charsets*.

Cela revient à dire que si vous (toujours en tant que navigateur) connaissez le charset dans lequel est codée la réponse, vous pouvez déterminer comment interpréter les octets que vous recevez. Une dernière question : comment déterminez-vous le charset ? Vous avez deux solutions. La première est de demander à l'utilisateur de vous indiquer le charset. Dans Netscape Navigator 4, cela se fait *via* Affichage ‹ Codage ; dans Netscape Navigator 6, c'est *via* Affichage ‹ Police de caractères. Dans Microsoft Internet Explorer 4, il faut utiliser Affichage ‹ Polices ; dans Microsoft Internet Explorer 5, c'est Affichage ‹ Codage. Cette approche impose souvent que l'utilisateur essaie plusieurs charsets avant que l'affichage ait un sens. La seconde possibilité est que le serveur (ou une servlet) indique le charset dans l'en-tête `Content-Type` que vous recevez. Par exemple, la valeur de `Content-Type` suivante :

```
text/html; charset=Shift_JIS
```

indique que le charset est Shift_JIS. Quelques navigateurs anciens peuvent être perturbés par l'inclusion d'un charset dans l'en-tête `Content-Type`.

Écrire une sortie codée

Maintenant que nous comprenons les charsets du point de vue du navigateur, il est temps de retourner à la servlet. Le rôle d'une servlet est de réaliser les choses suivantes :

1. Choisir un charset et l'utiliser pour la servlet.
2. Obtenir un `PrintWriter` pour ce charset.
3. Sortir les caractères qui peuvent être affichés à l'aide de ce charset.

L'*exemple 13-5* montre une servlet qui dit « Bonjour tout le monde » et affiche la date et l'heure courantes en japonais.

Exemple 13-5. Bonjour aux Japonais

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloJapan extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain; charset=Shift_JIS");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "ja");

    Locale locale = new Locale("ja", "");
    DateFormat full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                     DateFormat.LONG,
                                                     locale);
```

Exemple 13-5. Bonjour aux Japonais (suite)

```
    out.println("En japonais :");
    out.println("\u4eca\u65e5\u306f\u4e16\u754c");  // Bonjour tout le monde.
    out.println(full.format(new Date()));
  }
}
```

La *figure 13-4* montre une copie d'écran de l'*exemple 13-5*.

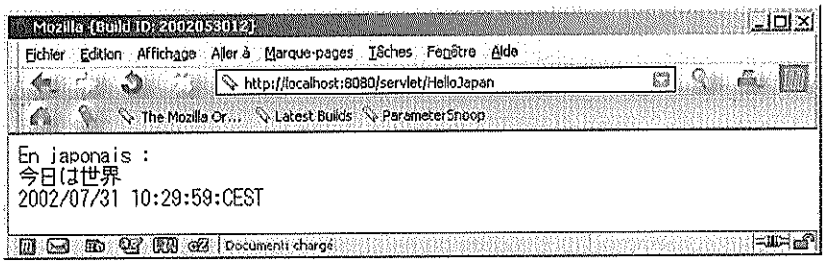


Figure 13-4. Bonjour en japonais

Cette servlet commence par définir le type de contenu à `text/plain` et le charset à `Shift_JIS`. Ensuite elle appelle `res.getWriter()` comme habituellement, excepté que dans ce cas le `PrintWriter` reçu est spécial. Il code la sortie de la servlet dans le charset Shift_JIS car c'est celui qui a été indiqué dans l'en-tête `Content-Type`. Cette deuxième ligne est donc équivalente au code suivant :

```
PrintWriter out = new PrintWriter(
  new OutputStreamWriter(res.getOutputStream(), "Shift_JIS"), true);
```

Notez que l'appel à `res.getWriter()` est susceptible de lancer une `UnsupportedEncodingException` si le charset n'est pas reconnu par Java[4] ou une `IllegalStateException` si `getOutputStream()` a déjà été appelée sur cette requête.

La servlet crée ensuite un `Locale` avec la langue `ja` pour représenter un environnement japonais générique et crée ensuite un `DateFormat`. Enfin, elle imprime l'équivalent de « Bonjour tout le monde » en japonais, en utilisant les échappements Unicode pour les caractères et sort la date et l'heure courante.

Pour que cette servlet fonctionne, le chemin des classes de votre serveur doit inclure les classes de conversion `sun.io.CharToByte*` ou leurs équivalents. Sur certaines plates-formes, elles ne sont pas toujours incluses par défaut. De même, pour que les glyphes japonais (ou ceux des autres langues) s'affichent correctement dans le navigateur, celui-ci doit supporter le charset et avoir accès aux polices de caractères nécessaires pour l'afficher.

Pour plus d'informations sur les possibilités d'internationalisation de Netscape Navigator, allez à *http://home.netscape.com/eng/intl/index.html* et à *http://www.microsoft.com/ie/intlhome.htm* pour Microsoft Internet Explorer.

---

4. Avec certaines versions précédentes de Java, elle peut, dans certains cas, lancer par erreur une `IllegalArgumentException` si le charset n'est pas reconnu.

Lire et écrire une sortie codée

Il est souvent extrêmement lent d'entrer des centaines ou des milliers d'échappements Unicode manuellement dans des fichiers source Java. Une solution plus facile est d'écrire la servlet à l'aide d'un éditeur de texte internationalisé et de sauvegarder le fichier dans le charset approprié. Tant que le charset est un de ceux reconnus par Java, la source peut être compilé avec presque n'importe quel compilateur Java moderne. Par exemple, voici comment demander au *javac* fourni avec le JDK de compiler un fichier de servlet codé en Shift_JIS :

```
javac -encoding Shift_JIS HelloJapan.java
```

Le fichier source *HelloJapan.java* doit être quasiment identique à celui de l'*exemple 13-5*, la seule différence étant que les échappements Unicode sont remplacés par des caractères japonais natifs. Si vous examiner le fichier codé avec un éditeur de texte japonais, vous verrez ces caractères entre les guillemets de `out.println()`. Si vous employez un autre éditeur de texte, ces caractères, et selon le codage tout le fichier, ressembleront à du charabia. En revanche, le contenu des fichiers *.class* est identique que la compilation se fasse avec les échappements Unicode ou les fichiers sources codés.

Une autre option, utile aux programmeurs qui ne comprennent la langue de la page affichée, consiste à écrire la servlet avec l'ASCII standard mais de lire le texte localisé à partir d'un fichier codé. Par exemple, supposons que le texte japonais correspondant à « Bonjour tout le monde » soit sauvegardé par une personne de l'équipe de localisation dans un fichier nommé *HelloWorld.ISO-2022-JP* et utilisant le codage ISO-2022-JP pour que les choses soient plus intéressantes. Une servlet peut lire ce fichier et en envoyer le contenu au navigateur en utilisant le codage Shift_JIS, comme montré à l'*exemple 13-6*.

Exemple 13-6. Envoyer une sortie localisée lue à partir d'un fichier

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloJapanReader extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain; charset=Shift_JIS");
    PrintWriter out = res.getWriter();
    res.setHeader("Content-Language", "ja");

    Locale locale = new Locale("ja", "");
    DateFormat full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                     DateFormat.LONG,
                                                     locale);
    out.println("En japonais :");

    try {
      FileInputStream fis = new FileInputStream(
        getServletContext().getRealPath("/HelloWorld.ISO-2022-JP"));
```

Exemple 13-6. Envoyer une sortie localisée lue à partir d'un fichier (suite)

```
      InputStreamReader isr = new InputStreamReader(fis, "ISO-2022-JP");
      BufferedReader reader = new BufferedReader(isr);
      String line = null;
      while ((line = reader.readLine()) != null) {
        out.println(line);
      }
    }
    catch (FileNotFoundException e) {
      // Pas de Bonjour pour vous.
    }

    out.println(full.format(new Date()));
  }
}
```

Cette servlet est principalement un convertisseur de codage de caractères. Elle lit le texte de *HelloWorld.ISO-2022-JP* codé en ISO-2022-JP et le convertit en interne en Unicode. Ensuite, elle sort le même texte converti de l'Unicode vers Shift_JIS.

Langues multiples

Il est temps maintenant d'aller un peu plus loin et d'essayer quelque chose qui n'est possible que depuis peu. Écrivons une servlet qui inclut plusieurs langues sur la même page. En un sens, nous avons déjà écrit une telle servlet. Notre précédent exemple, `HelloJapan`, incluait un texte en anglais et un texte en japonais. Cependant, il faut noter que c'est un cas spécial. Ajouter du texte anglais à une page est quasiment toujours possible, car presque tous les charsets incluent les 128 caractères de l'ASCII-US. Dans un cas plus général, lorsque le texte sur une page contient un mélange de langues et aucun des charsets précédemment mentionnés ne contient les caractères nécessaires, il nous faut employer une autre technique.

UCS-2 et UTF-8

La meilleure façon de générer une page contenant plusieurs langues et d'envoyer au client des caractères Unicode sur 16 bits. Il existe deux manières de procéder : UCS-2 et UTF-8. UCS-2 (*Universal Character Set*, forme sur 2 octets) envoie des caractères Unicode dans ce que l'on pourrait appeler leur forme naturelle, deux octets par caractères. Tous les caractères, y compris les caractères ASCII-US, sont sur 2 octets. UTF-8 (*UCS Transformation Format*, format sur 8 bits) est un codage à longueur variable. Avec UTF-8, un caractère Unicode est transformé en une représentation sur 1, 2 ou 3 octets. En général, UTF-8 est plus efficace qu'UCS-2 car il code un caractère de l'ASCII-US sur 1 octet. Pour cette raison, l'utilisation d'UTF-8 sur le web dépasse de loin celle de UCS-2. Pour plus d'informations sur UTF-8, reportez-vous à la RFC 2279 disponible à *http://www.ietf.org/rfc/rfc2279.txt*.

Avant de continuer, nous devons vous dire que le support de UTF-8 n'est pas encore garanti. Netscape a ajouté le support du codage UTF-8 pour la première fois dans Netscape Navigator 4 et Microsoft dans Internet Explorer 4.

Écrire en UTF-8

L'*exemple 13-7* montre comment une servlet qui utilise le codage UTF-8 dit « Bonjour tout le monde ! » et donne l'heure courante (dans le fuseau horaire local) en français, espagnol, japonais, chinois, coréen et russe.

Exemple 13-7. Une version servlet de la pierre de Rosette

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class HelloRosetta extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    Locale locale;
    DateFormat full;

    try {
      res.setContentType("text/plain; charset=UTF-8");
      PrintWriter out = res.getWriter();

      locale = new Locale("fr", "FR");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En français :");
      out.println("Bonjour tout le monde !");
      out.println(full.format(new Date()));
      out.println();

      locale = new Locale("es", "");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En Espa\u00f1ol:");
      out.println("\u00a1Hola Mundo!");
      out.println(full.format(new Date()));
      out.println();

      locale = new Locale("ja", "");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En japonais :");
```

Exemple 13-7. Une version servlet de la pierre de Rosette (suite)

```
      out.println("\u4eca\u65e5\u306f\u4e16\u754c");
      out.println(full.format(new Date()));
      out.println();

      locale = new Locale("zh", "");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En chinois :");
      out.println("\u4f60\u597d\u4e16\u754c");
      out.println(full.format(new Date()));
      out.println();

      locale = new Locale("ko", "");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En coréen :");
      out.println("\uc548\ub155\ud558\uc138\uc694\uc138\uacc4");
      out.println(full.format(new Date()));
      out.println();

      locale = new Locale("ru", "");
      full = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                            DateFormat.LONG,
                                            locale);
      out.println("En russe (cyrillique) :");
      out.print("\u0417\u0434\u0440\u0430\u0432\u0441\u0442");
      out.println("\u0432\u0443\u0439, \u041c\u0438\u0440");
      out.println(full.format(new Date()));
      out.println();
    }
    catch (Exception e) {
      log(ServletUtils.getStackTraceAsString(e));
    }
  }
}
```

La *figure 13-5* montre une copie d'écran de la sortie de la servlet.

Pour que cette servlet fonctionne, votre serveur doit supporter le JDK 1.1.6 ou ultérieur. Des versions précédentes de Java lançaient une `UnsupportedEncodingException` lorsque l'on essayait d'obtenir le `PrintWriter`, et la page restait blanche. Le problème provenait d'un alias de charset manquant. Java supporte le codage UTF-8 depuis la première version du JDK 1.1. Malheureusement, le JDK utilisait le nom `UTF8` pour le codage, alors que les navigateurs attendaient le nom `UTF-8`. Qui avait raison ? La réponse est arrivée début 1998, quand l'IANA (*Internet Assigned Numbers Authority*) a déclaré que `UTF-8` était la désignation préférée. (Allez à *http://www.isi.edu/in-notes/iana/assignments/character-sets*.) Peu après, le JDK 1.1.6 a ajouté `UTF-8` comme autre nom du codage

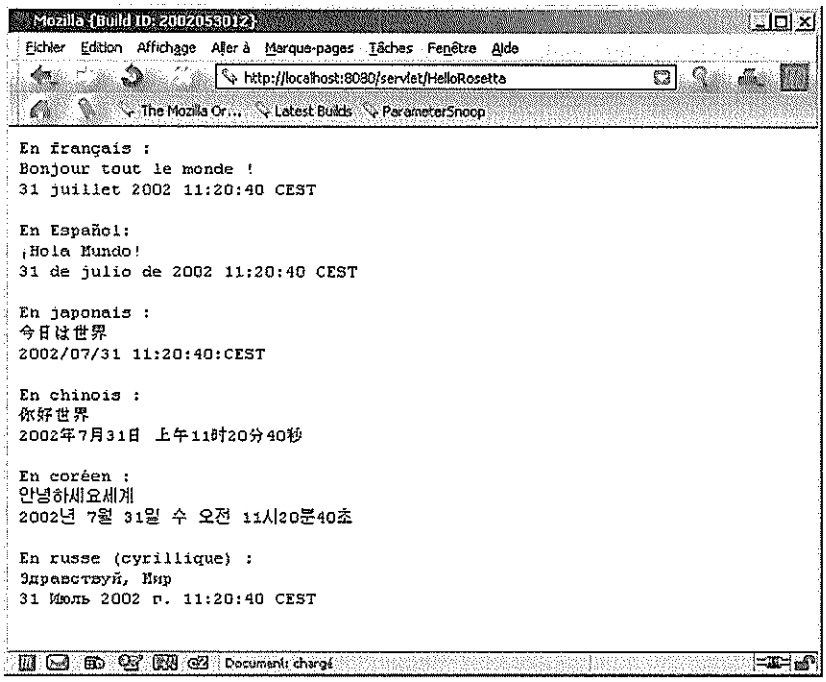


Figure 13-5. Un véritable Bonjour tout le monde

UTF8. Pour un maximum de portabilité sur les versions de Java, vous pouvez utiliser le nom UTF8 directement avec le code suivant :

```
res.setContentType("text/html; charset=UTF-8");
PrintWriter out = new PrintWriter(
  new OutputStreamWriter(res.getOutputStream(), "UTF8"), true);
```

De même, votre client doit supporter le codage UTF-8 et avoir accès à toutes les polices nécessaires. Sinon, une partie de la sortie ressemblera à du charabia.

Négociation dynamique de la langue

Allons encore un peu plus loin et examinons comment une servlet peut adapter sa sortie à la langue du client. Ainsi, la même URL peut livrer son contenu aux lecteurs à travers le monde dans leur langue maternelle.

Préférences concernant la langue

Une servlet a deux façons de connaître les préférences concernant la langue du client. Premièrement, le navigateur peut envoyer l'information dans la requête. Les navigateurs récents, comme Netscape Navigator 4 et Microsoft Internet Explorer 4, permettent aux utilisateurs de spécifier leur langue préférée. Dans Netscape Navigator 4 et 6, cela se passe dans Edition ‹ Préférences ‹ Navigateur ‹ Langues. Dans Microsoft Internet Explorer 4, il faut aller sous Affichage ‹ Options Internet ‹ Général ‹ Langues ; Internet Explorer 5 a placé les options Internet dans le menu Outils.

Un navigateur envoie les préférences de l'utilisateur sur la langue en utilisant l'en-tête HTTP `Accept-Language`. La valeur de cet en-tête précise la ou les langues que le client préfère. Notez que la spécification HTTP n'oblige pas à suivre ces préférences. Voici un exemple de valeur pour l'en-tête `Accept-Language` :

```
fr, en, es, de, ja, zh-TW
```

Cela indique que l'utilisateur client lit le français, l'anglais, l'espagnol, l'allemand, le japonais et le chinois adapté au taïwanais. Par convention, les langues sont données dans l'ordre de préférence. Chaque langue peut aussi inclure une *q-value* qui indique, sur une échelle de 0,0 à 1,0, une estimation des préférences de l'utilisateur pour cette langue. La q-value par défaut est 1,0 (préférence maximale). Voici un exemple de valeur pour l'en-tête `Accept-Language` avec des q-values :

```
fr, en, es;q=0.8, de;q=0.7, ja;q=0.3, zh-TW;q=0.1
```

Cette valeur d'en-tête signifie a peu près la même chose que l'exemple précédent.

La seconde façon pour une servlet de connaître les préférences de l'utilisateur est en les demandant. Par exemple, une servlet pourrait générer un formulaire demandant quelle langue l'utilisateur préfère. Par la suite, elle pourrait s'en souvenir et utiliser la réponse, peut-être en utilisant les techniques de suivi de session présentées au chapitre 7, *Suivi de session*.

Préférences concernant les charsets

En plus de l'en-tête HTTP `Accept-Language`, un navigateur peut envoyer un en-tête `Accept-Charset` qui indique au serveur les charsets qu'il comprend. Voici un exemple de valeur pour l'en-tête `Accept-Charset` :

```
iso-8859-1, utf-8
```

Cela indique que le navigateur comprend l'ISO-8859-1 et l'UTF-8. Si `Accept-Charset` n'est pas envoyé ou si sa valeur contient un astérisque (*), on peut supposer que les clients acceptent tous les charsets. Notez qu'aujourd'hui l'utilité de cet en-tête est limitée : peu de navigateurs envoient cet en-tête et ceux qui l'envoient ont tendance à envoyer un astérisque.

Lots de ressources

En utilisant `Accept-Language` (et, dans certains cas, `Accept-Charset`), une servlet peut déterminer la langue dans laquelle elle va parler à chaque client. Mais comment une servlet peut-elle gérer efficacement plusieurs versions localisées d'une page ? Une réponse est d'utiliser le support des lots (*bundles*) de ressources intégré à Java.

Un lot de ressources contient un ensemble de ressources localisées appropriées à une locale donnée. Par exemple, un lot de ressources pour une locale française pourrait contenir la traduction française de toutes les phrases utilisées par la servlet. Ensuite, lorsque la servlet détermine qu'elle veut parler français, elle charge ce lot de ressources et utilise les phrases stockées. Tous les lots de ressources étendent `java.util.ResourceBundle`. Une servlet charge un lot de ressources en utilisant la méthode statique `ResourceBundle.getBundle()` :

```
public static final
  ResourceBundle ResourceBundle.getBundle(String bundleName, Locale locale)
```

La servlet extrait les phrases du lot de ressources en utilisant la méthode `getString()` de `ResourceBundle` :

```
public final String ResourceBundle.getString(String key)
```

Un lot de ressources peut être créé de plusieurs manières. Pour les servlets, la technique la plus pratique consiste à mettre un fichier de propriétés spécial dans le chemin des classes du serveur qui contiendra les phrases traduites. Le nom du fichier doit se conformer au schéma *nomdelot_langue.properties* ou *nomdelot_langue_pays.properties*. Par exemple, utilisez *Messages_fr.properties* pour un lot français et *Messages_zh_TW.properties* pour un lot chinois/taïwanais. Le fichier contient les caractères ASCII-US dans le format suivant :

```
nom1=valeur1
nom2=valeur2
...
```

Chaque ligne peut aussi contenir des espaces et des échappements Unicode. L'information contenue dans ce fichier peut être chargée automatiquement par la méthode `getBundle()`.

Écrire de manière adaptée à chacun

L'*exemple 13-8* montre l'utilisation de `Accept-Language`, `Accept-Charset` et des lots de ressources dans une servlet qui dit « Bonjour tout le monde » à chaque client dans sa langue préférée. Voici un exemple de fichier de propriétés pour un lot de ressources en français, que vous pouvez stocker sous le nom *HelloBabel_fr.properties* dans un chemin utilisé par le chargeur de classes du serveur (comme dans *WEB-INF/classes*) :

```
greeting=Bonjour tout le monde
```

Et voici un lot de ressources pour le japonais, stocké dans le fichier *HelloBabel_ja.properties* :

```
greeting=\u4eca\u65e5\u306f\u4e16\u754c
```

Cette servlet `HelloBabel` utilise la classe `com.oreilly.servlet.LocaleNegotiator` qui contient la logique pour déterminer quel `Locale`, quel charset et quel `ResourceBundle` utiliser. Son code est montré à la section suivante.

Exemple 13-8. Une version servlet de la Tour de Babel

```
import java.io.*;
import java.util.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.LocaleNegotiator;
import com.oreilly.servlet.ServletUtils;

public class HelloBabel extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
```

Exemple 13-8. Une version servlet de la Tour de Babel (suite)

```
    try {
      String bundleName = "HelloBabel";
      String acceptLanguage = req.getHeader("Accept-Language");
      String acceptCharset = req.getHeader("Accept-Charset");

      LocaleNegotiator negotiator =
        new LocaleNegotiator(bundleName, acceptLanguage, acceptCharset);

      Locale locale = negotiator.getLocale();
      String charset = negotiator.getCharset();
      ResourceBundle bundle = negotiator.getBundle();  // Peut être null.

      res.setContentType("text/plain; charset=" + charset);
      res.setHeader("Content-Language", locale.getLanguage());
      res.setHeader("Vary", "Accept-Language");

      PrintWriter out = res.getWriter();

      DateFormat fmt = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                      DateFormat.LONG,
                                                      locale);
      if (bundle != null) {
        out.println("En " + locale.getDisplayLanguage() + " :");
        out.println(bundle.getString("greeting"));
        out.println(fmt.format(new Date()));
      }
      else {
        out.println("Impossible de trouver le lot.");
      }
    }
    catch (Exception e) {
      log(ServletUtils.getStackTraceAsString(e));
    }
  }
}
```

Cette servlet commence par définir le nom du lot qu'elle veut utiliser et retrouve ensuite les en-têtes `Accept-Language` et `Accept-Charset`. Elle crée un `LocaleNegotiator`, lui passant cette information, et lui demande quel `Locale`, quel charset et quel `ResourceBundle` il utilise. Notez qu'une servlet peut ignorer le charset retourné et utiliser le codage UTF-8. Rappelez-vous simplement qu'UTF-8 n'est pas aussi largement supporté que les charsets retournés par `LocaleNegotiator`. Ensuite, la servlet définit ses en-têtes : son en-tête `Content-Type` donne le charset, `Content-Language` la langue locale et `Vary` indique au client (si par chance il s'en préoccupe) que cette servlet peut faire varier sa sortie en fonction de l'en-tête `Accept-Language` du client.

Une fois les en-têtes définis, la servlet génère sa sortie. Elle commence par obtenir un `PrintWriter` qui correspond au charset. Ensuite elle dit, dans la langue par défaut, généralement le français, dans quelle langue on va souhaiter la bienvenue. Puis, elle re-

trouve et sort la phrase de bienvenue appropriée à partir du lot de ressources. Et enfin, elle imprime la date et l'heure appropriées à la locale du client. Si le lot de ressources est null, comme cela arrive quand le lot correspondant aux préférences du client n'existe pas, la servlet indique simplement qu'aucun lot n'a pu être trouvé.

La classe LocaleNegotiator

Le code de LocaleNegotiator est montré à l'*exemple 13-9*. Sa classe utilitaire, LocaleToCharsetMap, est montrée dans l'*exemple 13-10*. Si vous pouvez vous satisfaire de voir le négociateur de locale comme une boîte noire, vous pouvez sauter cette section.

LocaleNegotiator fonctionne en parcourant les préférences du client concernant la langue à la recherche de toute langue pour laquelle il existe un lot de ressources. Une fois une correspondance trouvée, elle utilise LocaleToCharsetMap pour déterminer le charset. S'il y a un problème, elle essaie de se rabattre sur l'anglais US. La logique ignore les préférences du client concernant le charset.

L'aspect le plus complexe du code de LocaleNegotiator est la gestion du comportement malencontreux de ResourceBundle.getBundle(). La méthode getBundle() essaie d'agir intelligemment. Si elle ne peut trouver un lot qui correspond exactement à la locale indiquée, elle essaie de trouver une équivalence proche. Le problème, pour nous, est que getBundle() considère que le lot de ressources pour la locale par défaut est une bonne équivalence. Donc, lorsque nous bouclons sur les langues du client, il est difficile de déterminer si nous avons une correspondance exacte ou non. La solution est de commencer par récupérer le lot de ressources de repli et ensuite d'utiliser cette référence pour déterminer lorsque l'on a une correspondance exacte. Cette logique est encapsulée dans la méthode getBundleNoFallback().

Exemple 13-9. La classe LocaleNegotiator

```
package com.oreilly.servlet;

import java.io.*;
import java.util.*;

import com.oreilly.servlet.LocaleToCharsetMap;

public class LocaleNegotiator {

  private ResourceBundle chosenBundle;
  private Locale chosenLocale;
  private String chosenCharset;

  public LocaleNegotiator(String bundleName,
                          String languages,
                          String charsets) {

    // Spécifie les valeurs par défaut :
    //   Langue Anglais, charset ISO-8859-1 (Latin-1), lot Anglais
    Locale defaultLocale = new Locale("en", "US");
```

Exemple 13-9. La classe LocaleNegotiator (suite)

```
String defaultCharset = "ISO-8859-1";
ResourceBundle defaultBundle = null;
try {
  defaultBundle = ResourceBundle.getBundle(bundleName, defaultLocale);
}
catch (MissingResourceException e) {
  // Pas de lot par défaut. Vol sans visibilité.
}

// Si le client ne spécifie pas une langue acceptable, nous pouvons
// conserver les valeurs par défaut.
if (languages == null) {
  chosenLocale = defaultLocale;
  chosenCharset = defaultCharset;
  chosenBundle = defaultBundle;
  return;  // Sortie rapide.
}

// Utiliser un tokenizer pour séparer les langues acceptables.
StringTokenizer tokenizer = new StringTokenizer(languages, ",");

while (tokenizer.hasMoreTokens()) {
  // Obtenir la prochaine langue acceptable.
  // (La langue peut ressembler à "fr; qvalue=0.91")
  String lang = tokenizer.nextToken();

  // Obtenir la locale pour cette langue.
  Locale loc = getLocaleForLanguage(lang);

  // Obtenir le lot pour cette locale. Ne pas laisser la recherche se
  // replier sur d'autres langues !
  ResourceBundle bundle = getBundleNoFallback(bundleName, loc);

  // Le lot retourné est null s'il n'y a pas de correspondance. Dans ce
  // cas, nous ne pouvons pas utiliser cette langue puisque la servlet ne
  // peut la parler.
  if (bundle == null) continue;  // Langue suivante.

  // Trouver un charset que nous pouvons utiliser pour cette locale.
  String charset = getCharsetForLocale(loc, charsets);

  // Le charset retourné est null s'il n'y a pas de correspondance. Nous
  // ne pouvons utiliser cette langue car la servlet ne peut la coder.
  if (charset == null) continue;  // Langue suivante.

  // Si nous arrivons là, il n'y a pas de problèmes avec la langue.
  chosenLocale = loc;
  chosenBundle = bundle;
```

Exemple 13-9. La classe LocaleNegotiator (suite)

```
      chosenCharset = charset;
      return;  // Terminé.
    }

    // Pas de correspondance, alors on conserve les valeurs par défaut.
    chosenLocale = defaultLocale;
    chosenCharset = defaultCharset;
    chosenBundle = defaultBundle;
  }

  public ResourceBundle getBundle() {
    return chosenBundle;
  }

  public Locale getLocale() {
    return chosenLocale;
  }

  public String getCharset() {
    return chosenCharset;
  }

  private Locale getLocaleForLanguage(String lang) {
    Locale loc;
    int semi, dash;

    // Enlever toute q-value qui pourrait venir après un point-virgule.
    if ((semi = lang.indexOf(';')) != -1) {
      lang = lang.substring(0, semi);
    }

    // Enlver les espaces avant et après.
    lang = lang.trim();

    // Créer un Locale à partir de cette langue. Un tiret peut séparer la
    // langue et le pays.
    if ((dash = lang.indexOf('-')) == -1) {
      loc = new Locale(lang, "");  // Pas de tiret, pas de pays.
    }
    else {
      loc = new Locale(lang.substring(0, dash), lang.substring(dash+1));
    }

    return loc;
  }

  private ResourceBundle getBundleNoFallback(String bundleName, Locale loc) {
```

Exemple 13-9. La classe LocaleNegotiator (suite)

```
    // D'abord récupérer le lot de repli, celui qui sera sélectionné si
    // getBundle() ne peut trouver de correspondace directe. Ce lot peut
    // être comparé aux lots retournés par des appels à getBundle() afin
    // de détecter quand getBundle() trouve une correspondance directe.
    ResourceBundle fallback = null;
    try {
      fallback =
        ResourceBundle.getBundle(bundleName, new Locale("bogus", ""));
    }
    catch (MissingResourceException e) {
      // Pas de lot de repli trouvé.
    }

    try {
      // Obtenir le lot de la locale spécifiée.
      ResourceBundle bundle = ResourceBundle.getBundle(bundleName, loc);

      // Le lot est-il différent du lot de repli ?
      if (bundle != fallback) {
        // Nous avons une vraie correspondance !
        return bundle;
      }
      // Ainsi le lot est le même que celui de repli. Nous pouvons encore
      // avoir une correspondance, mais seulement si notre langue de la locale
      // correspond à la langue de la locale par défaut.
      else if (bundle == fallback &&
            loc.getLanguage().equals(Locale.getDefault().getLanguage())) {
        // Une autre façon de trouver une correspondance.
        return bundle;
      }
      else {
        // Pas de correspondance, continuer à chercher.
      }
    }
    catch (MissingResourceException e) {
      // Pas de lot disponible pour cette locale.
    }

    return null;  // Pas de correspondance.
  }

  protected String getCharsetForLocale(Locale loc, String charsets) {
    // Note : Cette méthode ignore les charsets spécifiés.
    return LocaleToCharsetMap.getCharset(loc);
  }
}
```

Exemple 13-10. La classe LocaleToCharsetMap

```
package com.oreilly.servlet;

import java.util.*;

public class LocaleToCharsetMap {

  private static Hashtable map;

  static {
    map = new Hashtable();

    map.put("ar", "ISO-8859-6");
    map.put("be", "ISO-8859-5");
    map.put("bg", "ISO-8859-5");
    map.put("ca", "ISO-8859-1");
    map.put("cs", "ISO-8859-2");
    map.put("da", "ISO-8859-1");
    map.put("de", "ISO-8859-1");
    map.put("el", "ISO-8859-7");
    map.put("en", "ISO-8859-1");
    map.put("es", "ISO-8859-1");
    map.put("et", "ISO-8859-1");
    map.put("fi", "ISO-8859-1");
    map.put("fr", "ISO-8859-1");
    map.put("he", "ISO-8859-8");
    map.put("hr", "ISO-8859-2");
    map.put("hu", "ISO-8859-2");
    map.put("is", "ISO-8859-1");
    map.put("it", "ISO-8859-1");
    map.put("iw", "ISO-8859-8");
    map.put("ja", "Shift_JIS");
    map.put("ko", "EUC-KR");     // Nécessite le JDK 1.1.6.
    map.put("lt", "ISO-8859-2");
    map.put("lv", "ISO-8859-2");
    map.put("mk", "ISO-8859-5");
    map.put("nl", "ISO-8859-1");
    map.put("no", "ISO-8859-1");
    map.put("pl", "ISO-8859-2");
    map.put("pt", "ISO-8859-1");
    map.put("ro", "ISO-8859-2");
    map.put("ru", "ISO-8859-5");
    map.put("sh", "ISO-8859-5");
    map.put("sk", "ISO-8859-2");
    map.put("sl", "ISO-8859-2");
    map.put("sq", "ISO-8859-2");
    map.put("sr", "ISO-8859-5");
    map.put("sv", "ISO-8859-1");
    map.put("tr", "ISO-8859-9");
```

Exemple 13-10. La classe LocaleToCharsetMap (suite)

```
    map.put("uk", "ISO-8859-5");
    map.put("zh", "GB2312");
    map.put("zh_TW", "Big5");
  }

  public static String getCharset(Locale loc) {
    String charset;

    // Essayer une correspondance sur le nom complet (peut inclure le pays).
    charset = (String) map.get(loc.toString());
    if (charset != null) return charset;

    // Si le nom complet ne correspond pas, essayer juste la langue.
    charset = (String) map.get(loc.getLanguage());
    return charset;  // Peut être null.
  }
}
```

Locales fournies par le système

À partir de l'API Servlet 2.2, une servlet peut obtenir et fixer la locale préférée du client à l'aide de quelques méthodes utilitaires. `ServletRequest` offre une nouvelle méthode `getLocale()` qui retourne un objet `Locale` indiquant la locale préférée du client. Pour les servlet HTTP, la préférence repose sur l'en-tête `Accept-Language`. Il existe aussi une méthode `getLocales()` qui retourne un `Enumeration` d'objets `Locale` indiquant tous les locales acceptables pour le client, le préféré étant en premier. En plus de ces méthodes, la méthode `setLocale(Locale loc)` ajoutée à `ServletResponse` permet à une servlet de spécifier la locale de la réponse. La méthode définit automatiquement l'en-tête `Content-Language` et la valeur de charset `Content-Type`. La méthode `setLocale()` doit être appelée après `setContentType()` et avant `getWriter()` (car elle modifie le type de contenu et a un impact sur la création du `PrintWriter`). Par exemple :

```
public void doGet(HttpServletRequest req, httpServletResponse res)
            throws ServletException, IOException {
  res.setContentType("text/html");
  Locale locale = req.getLocale();
  res.setLocale(locale);
  PrintWriter out = res.getWriter();

  // Écrire la sortie basée sur locale.getLanguage().
}
```

Bien que ces méthodes conduisent à un code intéressant, elles n'apportent aucune assistance dans la détermination du support du `Locale` par l'application web. Pour cela, il faut placer une logique supplémentaire comme `LocaleNegotiator`.

Formulaires HTML

La gestion des formulaires HTML demande un peu de travail supplémentaire et quelques petites astuces si vous voulez utiliser un contenu localisé. Pour comprendre le problème, imaginons la situation suivante. Un formulaire HTML est envoyé dans une page en japonais. Il demande à l'utilisateur son nom, qu'il entre sous forme d'une chaîne de caractères japonais. Comment ce nom est-il soumis à la servlet ? Et, ce qui est plus important, comment la servlet le lit-elle ?

La réponse à la première question est que toutes les données d'un formulaire HTML sont envoyées sous forme d'une suite d'octets. Ces octets sont une représentation codée des caractères originaux. Avec les langues de l'Europe de l'ouest, le codage est celui par défaut, ISO-8859-1, avec un octet par caractère. Pour les autres langues, il peut y avoir d'autres codages. Les navigateurs ont tendance à coder les données du formulaire en utilisant le même codage que celui qui est appliqué à la page contenant le formulaire. Ainsi, si la page japonaise est codée en utilisant Shift_JIS, les données du formulaire envoyées seront aussi codées avec Shift_JIS. Cependant, notez que si la page ne spécifiait pas de charset et que l'utilisateur avait choisi manuellement le codage Shift_JIS pour l'affichage, plusieurs navigateurs auraient soumis les données du formulaire en utilisant l'ISO-8859-1[5]. En général, la chaîne d'octets codée contient un grand nombre d'octets spéciaux qui doivent être codés dans l'URL. Par exemple, si nous supposons que le formulaire japonais envoie le nom de l'utilisateur *via* une requête GET, l'URL résultante pourrait être la suivante :

http://serveur:port/servlet/NameHandler?name=%8CK%8C%B4%90%B3%8E%9F

La réponse à la seconde question, comment une servlet lit-elle les informations soumises, est un peu plus complexe. Une servlet a deux possibilités. Premièrement, elle peut laisser les données du formulaires telles quelles, les traitant comme une suite d'octets, chaque octet étant stocké comme un caractère dans la chaîne de paramètres. Cette tactique n'est utile qui si la servlet n'a pas besoin de manipuler les données et qu'elle peut être certaine que ces données ne sont envoyées qu'au même utilisateur en utilisant le même charset. Deuxièmement, une servlet peut convertir les données du formulaire de leur format natif codé en une chaîne de caractères Unicode. Cela permet à la servlet de manipuler librement le texte et de le sortir en utilisant d'autres charsets. Il y a cependant un problème. Les navigateurs ne fournissent pas aujourd'hui l'information indiquant le codage utilisé pour les données du formulaire. Ils pourraient fournir cette information dans le futur (probablement en utilisant l'en-tête `Content-Type` dans une requête POST). Mais pour le moment, c'est à la servlet de se débrouiller pour obtenir cette information.

Le charset caché

La technique couramment acceptée pour suivre le charset des données du formulaire soumises est d'utiliser un champ de formulaire caché pour le charset[6]. Sa valeur est le

5. Pour plus d'informations sur l'internationalisation de HTML ou des formulaires HTML, lisez la RFC 2070 à *http://www.ietf.org/rfc/rfc2070.txt*.

6. Les champs de formulaire cachés ont été présentés pour la première fois au chapitre 7 où nous les avons utilisés pour le suivi de session.

nom du charset de la page qui le contient. Ensuite, toute servlet recevant le formulaire peut lire la valeur de ce champ et savoir comment décoder les données envoyées.

L'*exemple 13-11* montre cette technique avec un générateur de formulaires qui définit le charset afin de correspondre à celui de la page. Voici un lot de ressources français qui pourrait accompagner la servlet, stocké dans *CharsetForm_fr.properties* :

```
title=Formulaire Charset
header=<H1>Formulaire Charset</H1>
prompt=Entrer le texte :
```

Et voici une ressource en japonais à stocker dans *CharsetForm_ja.properties* :

```
title=Formulaire Charset
header=<H1>\u6587\u5b57\u30bb\u30c3\u30c8\u30fb\u30d5\u30a9\u30fc\u30e0</H1>
prompt=\u30c6\u30ad\u30b9\u30c8\u3092\u5165\u529b\u3057\u3066\u304f\u3060\
\u3055\u3044
```

Exemple 13-11. Sauvegarder le charset dans un champ de formulaire caché

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.LocaleNegotiator;
import com.oreilly.servlet.ServletUtils;

public class CharsetForm extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    try {
      String bundleName = "CharsetForm";
      String acceptLanguage = req.getHeader("Accept-Language");
      String acceptCharset = req.getHeader("Accept-Charset");

      LocaleNegotiator negotiator =
        new LocaleNegotiator(bundleName, acceptLanguage, acceptCharset);

      Locale locale = negotiator.getLocale();
      String charset = negotiator.getCharset();
      ResourceBundle bundle = negotiator.getBundle();  // Peut être null.

      res.setContentType("text/html; charset=" + charset);
      res.setHeader("Content-Language", locale.getLanguage());
      res.setHeader("Vary", "Accept-Language");

      PrintWriter out = res.getWriter();

      if (bundle != null) {
        out.println("<HTML><HEAD><TITLE>");
        out.println(bundle.getString("title"));
```

Exemple 13-11. Sauvegarder le charset dans un champ de formulaire caché (suite)

```
        out.println("</TITLE></HEAD>");
        out.println("<BODY>");
        out.println(bundle.getString("header"));
        out.println("<FORM ACTION=CharsetAction METHOD=get>");
        out.println("<INPUT TYPE=hidden NAME=charset value=" + charset + ">");
        out.println(bundle.getString("prompt"));
        out.println("<INPUT TYPE=text NAME=text>");
        out.println("</FORM>");
        out.println("</BODY></HTML>");
      }
      else {
        out.println("Impossible de trouver le lot.");
      }
    }
    catch (Exception e) {
      log(ServletUtils.getStackTraceAsString(e));
    }
  }
}
```

Une copie d'écran de la version japonaise est montrée à la *figure 13-6*.

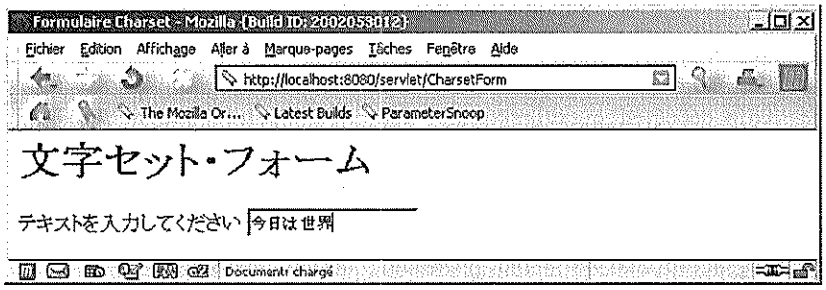


Figure 13-6. Un formulaire en japonais où l'utilisateur entre du texte

La servlet responsable du traitement du formulaire soumis est montrée à l'*exemple 13-12*. Elle lit le texte soumis et le convertit en Unicode, puis elle sort les caractères en utilisant le codage UTF-8. Elle affiche aussi la chaîne reçue sous forme d'échappements Unicode, montrant ce que vous auriez eu à mettre dans un fichier source Java ou un lot de ressources pour obtenir la même sortie. La servlet agit donc comme un traducteur de charset natif en une chaîne Unicode.

Exemple 13-12. Recevoir le charset dans un champ de formulaire caché

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Exemple 13-12. Recevoir le charset dans un champ de formulaire caché (suite)

```
public class CharsetAction extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    try {
      res.setContentType("text/plain; charset=UTF-8");
      PrintWriter out = res.getWriter();

      String charset = req.getParameter("charset");

      //  Obtenir le paramètre text.
      String text = req.getParameter("text");

      // Le convertir d'un tableau d'octets en un tableau de caractères.
      // Pour cela, utiliser le charset envoyé dans le champs caché.
      // Traiter la valeur originale comme des octets sur 8 bits placés dans
      // un String.
      text = new String(text.getBytes("ISO-8859-1"), charset);

      out.println("Charset reçu : " + charset);
      out.println("Texte reçu : " + text);
      out.println("Texte reçu (avec échappement) : " +
                  toUnicodeEscapeString(text));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    doGet(req, res);
  }

  private static String toUnicodeEscapeString(String str) {
    // D'après le code de java.util.Properties.save().
    StringBuffer buf = new StringBuffer();
    int len = str.length();
    char ch;
    for (int i = 0; i < len; i++) {
      ch = str.charAt(i);
      switch (ch) {
        case '\\': buf.append("\\\\"); break;
        case '\t': buf.append("\\t"); break;
        case '\n': buf.append("\\n"); break;
        case '\r': buf.append("\\r"); break;

        default:
```

Exemple 13-12. Recevoir le charset dans un champ de formulaire caché (suite)

```
            if (ch >= ' ' && ch <= 127) {
              buf.append(ch);
            }
            else {
              buf.append('\\');
              buf.append('u');
              buf.append(toHex((ch >> 12) & 0xF));
              buf.append(toHex((ch >>  8) & 0xF));
              buf.append(toHex((ch >>  4) & 0xF));
              buf.append(toHex((ch >>  0) & 0xF));
            }
          }
        }
        return buf.toString();
      }

      private static char toHex(int nibble) {
        return hexDigit[(nibble & 0xF)];
      }

      private static char[] hexDigit = {
        '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'
      };
    }
```

Un exemple de sortie est montré à la *figure 13-7*.

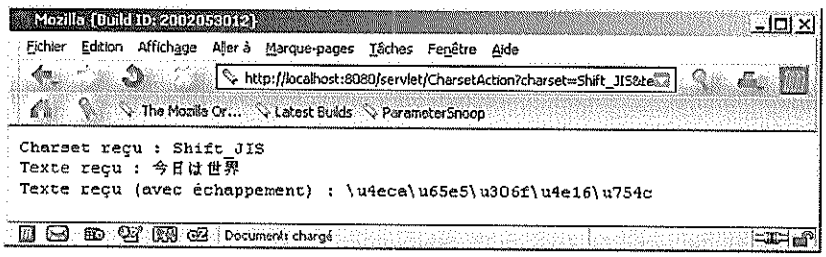


Figure 13-7. Traiter un formulaire en japonais

Le plus intéressant dans cette servlet est la petite partie qui reçoit et convertit le texte soumis :

```
text = new String(text.getBytes("ISO-8859-1"), charset);
```

L'appel `text.getBytes("ISO-8859-1")` convertit le texte en son format byte brut. Bien que la valeur du paramètre soit retournée comme un `String`, ce n'est pas un vrai `String`. Chaque caractère du `String` stocke en fait un octet du texte codé, nécessitant une conversion spéciale. Le constructeur de `String` englobant crée ensuite un `String` à partir des octets bruts en utilisant le codage spécifié par le champ `charset`. C'est affreux, mais cela fonctionne. Pour une solution plus élégante, il existe la classe `com.oreilly.servlet.ParameterParser` présentée au chapitre 19, *Affaires diverses*.

14

L'environnement Tea

Nous sommes prêts maintenant à présenter des environnements de création de contenu basé sur les servlets. Nous reportons la présentation des JSP car il s'agit de la solution de création de contenu la plus complexe et nous commençons par examiner Tea. TeaServlet (connu sous le nom *Tea*) est un produit du Walt Disney Internet Group (WDIG), précédemment GO.com, développé en interne au fil des ans afin de faciliter la création de sites web à fort trafic comme ESPN.com, NFL.com, Disney.com, Disney-Land.com, Movies.com, ABC.com et GO.com. Tea est devenu récemment en *open source* dans l'espoir que d'autres personnes le trouvent utile et continuent à l'améliorer. Cette stratégie a un sens ; vous partagez vos outils dans l'espoir que d'autres vous aident à les affûter. Nous présentons dans ce chapitre TeaServlet 1.1.0[1], disponible à *http://open-source.go.com*. Il nécessite l'API Servlet 2.1 ou 2.2 et le JDK 1.2 ou ultérieur. La licence de TeaServlet est du type Apache, une des licences les moins restrictives et des plus attrayantes pour les développeurs car elle vous permet d'utiliser Tea pour créer de nouveaux produits et de nouveaux sites sans en faire des versions en *open source*. C'est le modèle de licence utilisé par tous les projets Apache, y compris pour le serveur web Apache et le serveur Tomcat.

Tea a été conçu pour des projets mis en œuvre par de petites équipes de développeurs et de *producteurs techniques*. Un développeur crée les « applications » écrites en Java et installées dans TeaServlet. Un producteur crée et maintient l'apparence finale des pages web dynamiques en écrivant des modèles Tea qui appellent des fonctions fournies par les applications du développeur. Par exemple, sur le site web ESPN, un développeur crée une application pour les statistiques des équipes, un développeur (travaillant de façon indépendante) crée une application pour les statistiques d'un joueur, et un producteur technique poste les données sur le Web en utilisant les modèles Tea. En fait, plusieurs producteurs travaillant de façon indépendante peuvent employer les mêmes données pour créer des sites destinés à des publics différents, quelque chose que WDIG a fait pour ses sites, comme Disney.com, Movies.com, ESPN.com et GO.com. Les mo-

1. *N.d.T.* : au moment de la traduction de ce livre, la version courante de TeaServlet est la version 1.4.3.

dèles sont écrits dans le langage Tea afin de garantir une très bonne séparation du contenu et de la présentation.

Le langage Tea

Tea est un langage de programmation conçu pour la mise en forme du texte. Il est fortement typé, compilé et conçu pour fonctionner dans un environnement d'hébergement basé sur Java. TeaServlet constitue cet environnement d'hébergement, employant Tea pour la création de page web et fournissant un point d'ancrage standard dans les moteurs de servlets avec lequel on peut appeler les modèles de Tea.[2]

L'objectif principal du langage Tea est d'assurer la séparation entre le contenu et la présentation sans sacrifier aux constructions de base de la programmation. Dans ce but, Tea a été créé avec quatre contraintes de base :

- les données ou les types de données ne peuvent être créés directement ; ils doivent être acquis ;
- les données acquises ne peuvent être modifiées directement ;
- un modèle ne peut provoquer directement des dommages à son environnement d'hébergement ;
- seul un minimum d'instructions de programmation est fourni.

Ces contraintes protègent aussi le système des modèles bogués ou mal écrits, quelque chose d'indispensable quand votre site accepte des dizaines de nouveaux modèles, comme c'est le cas avec les sites d'informations. Cette conception profite aussi aux sociétés d'hébergement web qui ont besoin de fournir aux clients une possibilité de création de pages web dynamiques mais qui ne veulent pas que les erreurs d'un client nuisent au reste du serveur.

Mais avons-nous besoin d'un autre langage, spécialement un langage qui en fait *moins* que Java ? Le *Tea Template Language Manual* répond à cette question :

> Tea résulte de plusieurs années d'expérience avec d'autres mécanismes de construction de pages web. La plupart des applications de type web démarrent par des balises HTML incluses dans du code, qu'il soit en C, Perl ou Java. Cette approche est adaptée aux petits projets ou aux prototypes car le temps de développement est court.
>
> Les modifications de la mise en forme des pages peuvent être fréquentes, mais les développeurs ne souhaitent pas effectuer ces changements. Inévitablement, ils passent donc à une forme de mécanisme de modèles autorisant le remplacement d'éléments. Chaque élément n'est qu'un emplacement pour une chaîne de caractères contenant des données créées par l'application. Ces systèmes de modèles évoluent ensuite vers le support d'instructions spéciales pour la mise en forme de tableaux, de formulaires et une logique de conditions simple.
>
> Quand on introduit des instructions de programmation dans un modèle, le défi est

2. Bien que Tea ait été conçu pour la mise en forme du texte, il est possible qu'un modèle Tea contrôle une classe de production d'images et crée une image dynamique. Contrairement aux autres outils de création de contenu que nous verrons, les modèles Tea peuvent envoyer des données de type binaire et caractère.

d'obtenir quelque chose de suffisamment puissant mais dans le même temps simple et sûr. S'il est trop puissant, des applications complètes peuvent être développées dans des modèles. S'il est trop pauvre, le formatage HTML finit dans l'application. S'il n'est ni simple ni sûr, les développeurs d'applications finissent par écrire et maintenir les modèles.

Au lieu d'embarquer un langage existant dans quelque chose comme les ASP ou les JSP, Tea est un langage spécialement conçu pour satisfaire les contraintes d'un système de modèles. Il est sûr, simple, efficace et puissant.

Tea est intégré à une servlet spéciale. Cette servlet donne aux modèles Tea un contrôle sur la construction de la page, tout en évitant un lien fort à une application serveur écrite par un développeur Java. Alors que cette servlet fournit des fonctionnalités similaires à celles des JSP, Tea assure une bonne séparation du modèle et de la vue grâce aux limitations voulues du langage. Bien que cette séparation soit aussi suggérée dans les JSP, elle ne peut être garantie. De plus, les modèles Tea ne supportent pas les instructions de programmation qui pourraient être utilisées de manière irresponsable. Contrairement aux JSP, les modifications n'ont pas besoin de passer par une vérification stricte ou une phase de test.

Toutes les personnes travaillant sur un projet doivent effectuer leurs tâches le plus efficacement possible, et Tea apporte son aide en vous laissant faire exactement ce dont vous avez besoin, aussi facilement que possible, rien de plus. Même dans le cas de projets exécutés uniquement par des développeurs, l'emploi de Tea reste bénéfique. Il encourage de bonnes pratiques de développement et il rend les applications plus faciles à maintenir.

Les lecteurs de ce livre doivent aussi se rappeler que Tea n'est pas un langage qui vous est destiné ; c'est un langage utilisé par les producteurs techniques. C'est un langage plus simple, plus sûr et aussi efficace que Java.

Démarrage

Pour se familiariser avec Tea, commençons par examiner quelques modèles autonomes simples. Ces modèles n'exploitent aucune classe Java provenant d'une « application », ils ne sont donc pas capables de faire beaucoup de choses. Voici notre premier modèle :

```
<% template SimplePage() %>
Voici une page simple qui ne fait rien de spécial.
```

Ce modèle affiche simplement « Voici une page simple qui ne fait rien de spécial. » à toute personne qui y accède. Les modèles sont constitués de régions de code et de régions de texte. Les régions de code sont délimitées par <% et %> (aucun autre délimiteur spécial n'est requis). Le texte en dehors d'une région de code est sorti par le modèle tel quel. C'est pour cela que ce modèle affiche cette simple ligne.

Pour exécuter le modèle, vous devez d'abord le sauvegarder dans un fichier nommé *SimplePage.tea*. Comme pour les classes Java, le nom du fichier doit correspondre au nom du modèle déclaré par le constructeur (sensible à la casse) et le fichier doit se terminer par l'extension *.tea*. Le répertoire dans lequel placer les modèles est configurable ; nous recommandons un répertoire sous *WEB-INF*, comme *WEB-INF/templates*.

L'étape suivante pour exécuter le modèle est d'installer Tea lui-même. Téléchargez la distribution à partir de *http://opensource.go.com* et suivez leurs instructions

d'installation : placez le fichier *TeaServlet.jar* de la distribution dans le chemin de classes du serveur web. Modifiez ensuite le descripteur de déploiement *web.xml* de votre application afin d'enregistrer TeaServlet sous le nom tea avec un paramètre d'initialisation indiquant un emplacement pour les informations de configuration. Définissez aussi une règle de correspondance de préfixe afin que /tea/* invoque la servlet tea. Un exemple d'ajout à *web.xml* est montré à l'*exemple 14-1*.

Exemple 14-1. Installer TeaServlet

```
<!-- ... -->
<servlet>
    <servlet-name>
        tea
    </servlet-name>
    <servlet-class>
        com.go.teaservlet.TeaServlet
    </servlet-class>
    <init-param>
        <param-name>
            properties.file
        </param-name>
        <param-value>
            <!-- Modifiez ceci afin de donner un chemin absolu pour
                 votre fichier de propriétés -->
            /tomcat/webapps/teatime/WEB-INF/TeaServlet.properties
        </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>
        tea
    </servlet-name>
    <url-pattern>
        /tea/*
    </url-pattern>
</servlet-mapping>
<!-- ... -->
```

La seule chose à personnaliser est le chemin d'accès au fichier *TeaServlet.properties*. Ce doit être un chemin absolu. Le fichier *TeaServlet.properties* configure le comportement de TeaServlet. Il peut être assez long. C'est pourquoi le fichier *web.xml* pointe vers un fichier externe au lieu d'inclure directement les informations de configuration. Nous reportons la présentation complète du fichier de propriétés et utilisons le fichier *TeaServlet.properties* le plus simple possible :

```
# TeaServlet.properties
#
# Indiquer l'emplacement des modèles Tea.
template.path = /tomcat/webapps/teatime/WEB-INF/templates
```

Assurez-vous de placer le fichier de propriétés à l'endroit indiqué dans le fichier

web.xml. La dernière étape consiste à redémarrer votre serveur afin qu'il lise le nouveau fichier *web.xml* (avec certains serveurs, ce n'est pas nécessaire).

Une fois `TeaServlet` correctement installé, il est possible d'accéder à `SimplePage` par l'URL *http://localhost:8080/teatime/tea/SimplePage*. La partie */teatime* du chemin indique l'application web concernée. La partie */tea* est la correspondance de préfixe donnée dans le fichier *web.xml* de `teatime`. Elle provoque l'invocation de `TeaServlet` afin de gérer la requête. Les modèles peuvent être placés dans des sous-répertoires de `template.path`, auquel cas ils sont invoqués à l'aide d'un chemin qui inclut le nom du sous-répertoire comme dans */teatime/tea/sous-répertoire/SimplePage*.

Informations de requête

Créer un contenu statique demande beaucoup de travail, créons donc quelque de plus dynamique. Le constructeur de modèle peut accepter un nombre quelconque de paramètres dont les valeurs sont prises automatiquement dans les paramètres de requête (au travers de la chaîne d'interrogation et des données POST). Les paramètres peuvent être de type `String`, `Integer`, `Float`, `Double` et toute autre sous-classe de `java.lang.Number`. La conversion en types numériques se fait automatiquement. Si aucun paramètre d'un nom donné n'est envoyé ou si la conversion de type échoue, le modèle reçoit la valeur `null` pour ce paramètre. Le modèle ci-dessous utilise son constructeur pour accepter un paramètre appelé `name` :

```
<% template SimplePage(String name) %>
Voici une page simple qui ne fait rien de spécial,
excepté afficher un nom : <% name %>
```

Si vous accédez au modèle par */teatime/tea/SimplePage?name=Waldo*, il affiche « Waldo » sur la page. Si l'accès au modèle se fait sans donner de paramètre `name`, il affiche « null » sur la page. L'*exemple 14-2* montre un modèle qui vérifie ce cas.

Exemple 14-2. Un modèle pour vérifier les noms

```
<% template SimplePage(String name) %>
Voici une page simple qui ne fait rien de spécial,
excepté afficher un nom : <% name %>
<P>

<%
if (name == null) {
  "Vous n'avez pas donné de nom"
}
if (name == "Slim Shady") {
  "Veuillez vous lever"
}
%>
```

Ce nouveau modèle vérifie si le nom est `null` et, si c'est le cas, affiche un avertissement à l'utilisateur. Il vérifie aussi si le nom est `Slim Shady` et, si c'est le cas, traite de façon spéciale l'utilisateur. Il est important de noter deux choses sur la comparaison avec `Slim Shady` : premièrement, les valeurs de type chaîne dans Tea peuvent être comparées *via*

l'opérateur == (comme tous les objets). Cela signifie qu'il n'est pas nécessaire d'expliquer au producteur technique le but de la méthode `.equals()`. Deuxièmement, une chaîne `null` peut être comparée à une autre chaîne sans risquer d'obtenir une `NullPointerException`, quelque chose qui peut se produire avec `.equals()`.

Aller plus loin

Avec TeaServlet, d'autres informations de requête sont disponibles. Par exemple, un modèle Tea peut accéder directement aux en-têtes de requête et aux cookies, ainsi qu'aux paramètres. L'*exemple 14-3* montre un modèle Tea qui analyse les informations de requête. En réalité, il est plutôt rare qu'un modèle Tea accède à ces informations directement. Il laissera plutôt l'application fournir une version traitée des informations à afficher.

Exemple 14-3. Un modèle pour analyser les informations de requête

```
<% template Snoop() %>

<HTML><HEAD><TITLE>Analysons !</TITLE></HEAD><BODY>

<% request = getRequest()  // Le type de l'objet est implicite.
%>

<H1>Informations diverses</H1>
Chaîne d'interrogation : <% request.queryString %> <BR>
Utilisateur distant : <% request.remoteUser %> <BR>

<H1>Informations sur les paramètres</H1>
<%
foreach (paramName in request.parameters.names) {
  paramName " : " request.parameters[paramName] "<BR>"
}
%>

<H1>Informations sur les attributs de requête</H1>
<%
foreach (attribName in request.attributes.names) {
  attribName ' : ' request.attributes[attribName] '<BR>'
}
%>


<H1>Informations sur les en-têtes</H1>
<%
foreach (headerName in request.headers.names) {
  headerName ' : ' request.headers[headerName] '<BR>'
}
%>

<h1>Informations sur les cookies</h1>
```

Exemple 14-3. Un modèle pour analyser les informations de requête (suite)

```
<%
foreach (cookie in request.cookies.all) {
  cookie.name ' : ' cookie.value '<BR>'
}
%>
La valeur du cookie de l'identifiant de session est
  <% request.cookies["JSESSIONID"].value %>.

<% // Des choses que l'on ne peut pas faire en Tea...
%>
<%/*
"<H1>Lecture à partir de la session de l'utilisateur</H1>"
request.session["peut"] = "fonctionner plus tard"

"<H1>Définir un attribut</H1>"
request.attributes["ne"] = "fonctionnera pas"

"<H1>Définir un cookie</H1>"
// La manipulation de la réponse se fait par des fonctions de application.
*/%>

</BODY></HTML>
```

Nous pouvons apprendre beaucoup sur Tea à partir de cet exemple simple. Premièrement, les variables locales (comme `request`) n'ont pas besoin d'être déclarée avec un type d'objet. Elles sont fortement typées, mais l'affectation du type est implicite. Si nécessaire, le langage propose le mot-clé `isa` pour gérer le transtypage.

Nous utilisons une instruction `foreach` qui permet de parcourir un ensemble d'objets. Elle est très souple. Elle est capable d'itérer sur des tableaux comme sur tout objet implémentant `java.util.Collection`. L'instruction `foreach` supporte aussi l'itération sur un intervalle et l'itération en sens inverse. Par exemple :

```
foreach (compteur in 1 .. 10 reverse) { compteur '<BR>' }
```

affiche un compte à rebours débutant à 10. L'instruction `foreach` traite tous les aspects de l'itération dans Tea ; il n'y a pas d'instruction `for` ou `while`. (Ainsi, il n'y a pas non plus de boucle infinie dans Tea.)

Les instructions Tea n'ont pas besoin de se terminer par un point-virgule comme en Java. Les instructions séparées par un passage à la ligne n'existent pas non plus. À la place, le compilateur Tea exploite la grammaire Tea très soigneusement conçue pour déterminer la fin des instructions ! Les points-virgules sont néanmoins supportés pour le rare cas où il est nécessaire de lever une ambiguïté dans la séparation des instructions et pour les programmeurs Java qui les ajoutent instinctivement.

La plupart du temps, on n'a pas besoin des opérateurs de concaténation de chaînes dans Tea. La présence de chaînes entourées de guillemets doubles (`""`) ou simples (`''`) dans un bloc de code Tea fournit généralement assez d'informations pour déterminer qu'il doit y avoir une concaténation de chaînes. Pour une concaténation plus explicite, utilisez le caractère `&`. Le caractère `+` n'est pas employé car la surcharge de l'opérateur `+` conduit à des ambiguïtés de typage.

Dans Tea, les commentaires suivent la norme de Java, où /* et */ dénotent un commentaire sur plusieurs lignes et // débute un commentaire sur une seule ligne. Ces commentaires ne sont pas envoyés au client, tant que vous les placez dans un bloc de code Tea.

Nous avons placé un commentaire multiligne pour commenter le dernier bloc de code montrant ce qu'il est impossible de faire : lire à partir de la session de l'utilisateur, définir un attribut de requête, définir un cookie et, en fait, définir quelque chose dans la session. La possibilité d'accéder à la session pourra être ajoutée plus tard, par les contributions en *open source* à Tea ; elle n'est pas supportée actuellement uniquement parce que WDIG utilise en interne un autre mécanisme de suivi de session. Les autres possibilités sont volontairement non supportées et doivent être réalisées par du code Java dans l'application support, du fait de la contrainte de conception stipulant que « Les données acquises ne peuvent être modifiées directement. ».

Administration de Tea

L'environnement TeaServlet fournit une page web d'administration centralisée (écrite en Tea, bien entendu) avec laquelle l'administrateur du site peut gérer tous les modèles Tea et les classes de l'application. L'écran d'administration permet de voir la liste des modèles actuellement chargés, de voir la liste des applications activées ainsi que les fonctions qu'elles rendent disponibles, dans un style Javadoc. Il fournit aussi un accès aux journaux d'exécution de modèles, permettant à l'administrateur de voir tous les journaux ou de sélectionner un modèle et un type de journalisation (debug, `info`, `warn` ou `error`). L'écran d'administration apporte aussi un mécanisme de contrôle du rechargement des modèles, comme nous le verrons en détail plus loin. Une copie d'écran de la page Templates est montrée à la *figure 14-1*.

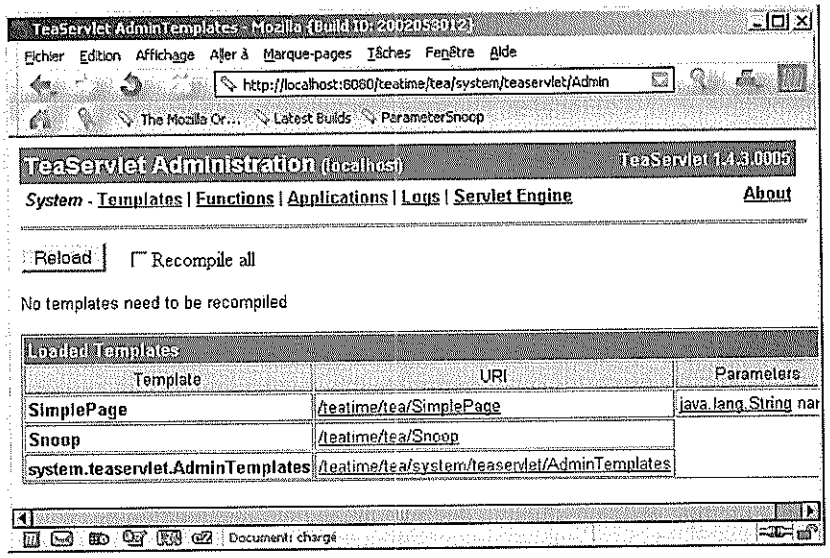


Figure 14-1. Administration des modèles actuellement chargés

Pour exécuter l'application d'administration de TeaServlet, vous devez d'abord configurer le fichier *TeaServlet.properties*. Un exemple est donné à l'*exemple 14-4*..

Exemple 14-4. Un premier fichier TeaServlet.properties

```
# TeaServlet.properties

# Indiquer l'emplacement des modèles Tea (les fichiers .tea).
template.path = /tomcat/webapps/teatime/WEB-INF/templates

# Spécifier un chemin optionnel dans lequel le système écrit les modèles
# compilés, sous forme de fichiers .class.
template.classes = /tomcat/webapps/teatime/WEB-INF/templateClasses

# Spécifier un modèle par défaut optionnel chargé si aucun n'est indiqué.
template.default = Index

# Spécifier les applications support à charger dans le système de TeaServlet.
applications {
    "System" {
        # SystemApplication fournit le support de l'administration de
        # TeaServlet.
        class = com.go.teaservlet.AdminApplication
        init {
            # La clé de sécurité pour la page d'administration.
            admin.key = admin
            admin.value = true
        }
    }
    "Autre" {
        class = AutresApplicationsEnsuite
    }
}

# Spécifier les messages de l'application affichés dans les journaux.
log.debug = true
log.info = true
log.warn = true
log.error = true
```

Le fichier *TeaServlet.properties* configure le système de TeaServlet tout comme le fichier *web.xml* configure une application web. Ce fichier indique à `TeaServlet` où trouver les fichiers de modèles, où placer les classes de modèles compilées afin qu'elles puissent être sauvegardées entre les redémarrages du serveur, quelles applications doivent être chargées pour supporter les modèles et quels types de messages de journalisation doivent être écrits par défaut. Chaque application doit être enregistrée dans *TeaServlet.properties* avec un fichier de classe indiqué et des paires optionnelles de nom/valeur servant de paramètres d'initialisation.

La structure de *TeaServlet.properties* est similaire au fichier `java.util.Properties` traditionnel avec plusieurs améliorations importantes : l'ordre des éléments est préservé, les

guillemets (simples ou double) peuvent être utilisés pour définir des clés comportant des espaces et les propriétés peuvent être imbriquées à l'aide de parenthèses. L'*exemple 14-4* pourrait être écrit en utilisant un fichier de propriétés Java normal comme le montre le code suivant (excepté que sa lecture par la classe `java.util.Properties` ne conserve pas l'ordre des éléments). Le fichier aurait pu être écrit en XML, même si pour des utilisations aussi simples que celle-ci, XML est sans doute un peu exagéré :

```
template.path=/tomcat/webapps/teatime/WEB-INF/templates
template.classes=/tomcat/webapps/teatime/WEB-INF/templatesClasses
applications.System.class=com.go.teaservlet.AdminApplication
applications.System.init.admin.key=admin
applications.System.init.admin.value=true
applications.Other.class=MoreOnApplicationsLater
log.debug=true
log.info=true
log.warn=true
log.error=true
```

Après avoir édité le fichier *TeaServlet.properties* et redémarré votre serveur, le système est prêt à être administré. L'emplacement par défaut de l'application d'administration (en supposant qu'elle est installée dans le contexte */teatime*) est *http://localhost:8080/teatime/tea/system/teaservlet/Admin?admin=true*. *?admin=true* fonctionne comme une précaution de sécurité primitive ; le nom et la valeur doivent correspondre à la paire `admin.key`/`admin.value` donnée dans le fichier *TeaServlet.properties*. Une personne extérieure ne trouvera vraiment rien d'important dans l'accès à la page d'administration, mais avoir une porte grande ouverte n'étant pas non plus très prudent, ce verrou primitif a été ajouté. La communauté des contributeurs ne devrait être longue à proposer un mécanisme de sécurité amélioré.

L'utilisation la plus courante de l'application d'administration est le rechargement des fichiers de modèle. Les modèles ne sont pas rechargés automatiquement lors des accès comme le sont les pages JSP. Les modifications apportées aux modèles ne sont pas vues avant d'avoir cliqué sur le bouton Reload Changes de l'écran d'administration des modèles. La compilation des modèles uniquement à la demande est plus efficace que de contrôler l'estampille d'un fichier à chaque requête et elle crée aussi une étape de publication plus explicite pour le site. L'application d'administration rejette toutes les modifications si la compilation d'un modèle échoue, permettant ainsi à la version précédente de persister afin que les clients ne puissent *absolument pas* voir une erreur de compilation. Toutes les erreurs survenant pendant la compilation sont listées dans l'application d'administration, comme le montre la *figure 14-2*.

Notez comment l'emplacement et la description de l'erreur sont précis. TeaServlet arrive à cela car les modèles Tea se compilent directement dans du bytecode Java ! Aucun fichier *.java* intermédiaire n'est créé et le compilateur Java n'est pas nécessaire. La compilation directe des modèles en bytecode exploite les technologies JIT et HotSpot afin de maximiser les performances. Mais contrairement aux pages JSP, le compilateur fonctionne directement sur les sources écrites par un humain et peut ainsi mieux diagnostiquer les erreurs sans le fichier intermédiaire qui ajoute un niveau d'indirection supplémentaire. De même, Tea ne nécessite pas de livrer un compilateur Java avec le serveur web comme c'est le cas pour les JSP, une contrainte pleine de difficultés avec les licences. Tea peut aussi précompiler les modèles, permettant la distribution de modèles sans leur source.

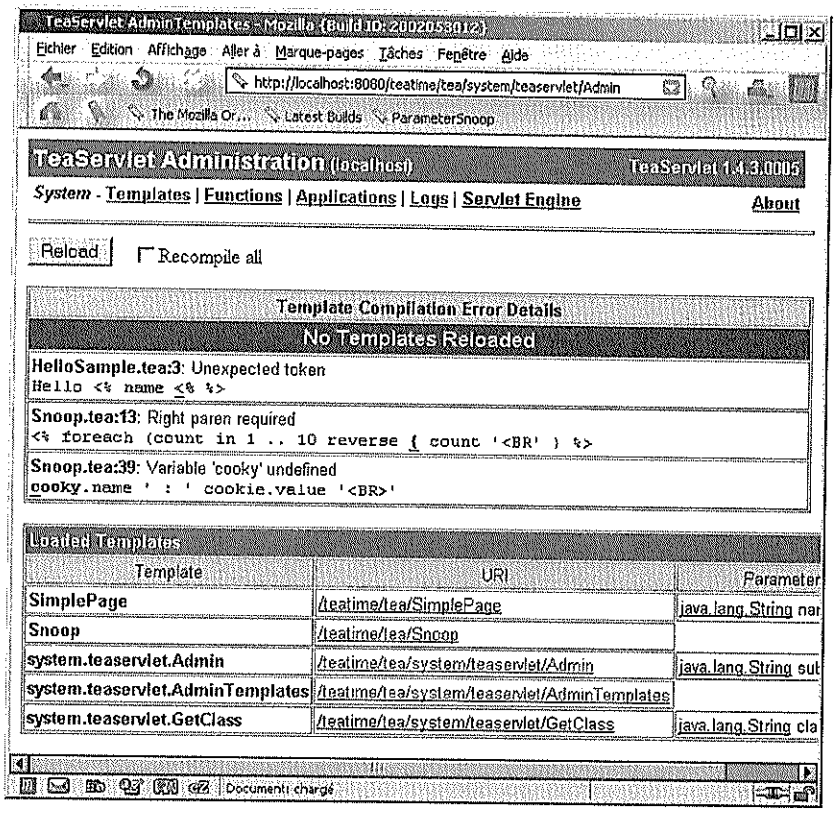


Figure 14-2. Messages d'erreur de compilations complets et précis

Modèles compilés : une opportunité commerciale

La possibilité de distribuer des modèles Tea précompilés ouvre un nouveau modèle commercial intéressant. Un fournisseur créant une application web à l'aide de Tea a le choix de vendre l'application web sans donner le code source original des modèles. Seule la bibliothèque Tea est nécessaire. Elle est portable sur les serveurs et peut venir avec l'application. Être capable de livrer un produit sans son code peut encourager certains fournisseurs à produire de nouvelles applications web et à les vendre.

Notez que les JSP supportent aussi les pages précompilées. Malheureusement, le bytecode généré par les JSP dépend souvent des classes spécifiques au conteneur et les classes générées doivent suivre une convention de nommage spécifique au conteneur (voir l'*exemple 18-2* du chapitre 18, *JavaServer Pages*). Les pages JSP précompilées ont donc tendance à être liées à un conteneur particulier.

Tea gère aussi élégamment les erreurs d'exécution. Les exceptions lancées au sein de Tea contiennent des traces de la pile qui incluent les numéros de ligne du modèle :

```
2000/07/10 19:29:12.105 PDT> java.lang.NullPointerException:
    at com.go.teaservlet.template.SimplePage.execute(SimplePage.tea:5)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.go.tea.runtime.TemplateLoader$TemplateImpl.execute(
    TemplateLoader.java, Compiled Code)
    at com.go.teaservlet.TeaServlet.processTemplate(TeaServlet.java,
    Compiled Code)
    at com.go.teaservlet.TeaServlet.doGet(TeaServlet.java:238)
    <etc>
```

Ces messages sont visibles dans l'application d'administration de TeaServlet. Quand ils surviennent, le client reçoit par défaut une page d'erreur avec le code d'état 500, sans trace de la pile pour des raisons de sécurité.

Tea dispose aussi d'une fonctionnalité de « gardien d'exceptions ». Elle permet à Tea de continuer le traitement d'une page tout en journalisant l'exception. Tea affiche toutes les parties de la page qu'il peut traiter et omet la petite portion affectée par l'exception. Pour beaucoup de sites, une page dont il manque une petite partie est préférable à une erreur du serveur retournée à l'utilisateur.

Applications Tea

Jusqu'à présent, nous avons examiné des modèles Tea autonomes très pauvres (intentionnellement). Mais les modèles Tea sont comme les soldats du front dans l'infanterie. Seuls ils sont très faibles, mais secondés par un support aérien conséquent, ils deviennent extrêmement puissants. Voici quelques façons de rendre les modèles Tea plus puissants.

Les modèles peuvent accéder à toutes les fonctions de n'importe quelle « application » installée dans TeaServlet par le biais du fichier *TeaServlet.properties*. (Chaque application web peut avoir un fichier *TeaServlet.properties* différent et donc un ensemble d'applications installées différent.) Les modèles appellent les fonctions en spécifiant, dans un bloc de code Tea, le nom de la fonction suivi de parenthèses ouvrantes et fermantes. Les fonctions acceptent des paramètres et retournent des valeurs :

```
<% list = getRecords() %>
<% removeRecord(8086) %>
```

L'écran d'administration de TeaServlet montre une liste de toutes les fonctions disponibles à partir des applications installées. Si deux applications déclarent la même fonction, le modèle peut préfixer l'appel de la fonction par le nom de l'application afin de lever l'ambiguïté :

```
<% list = RecordListApp.getRecords() %>
```

Il existe un ensemble de fonctions utilitaires disponibles à tous les modèles et conçues de manière à faciliter les traitements de base du texte. Les sections suivantes listent certaines des fonctions utilitaires les plus employées.

Traitement du texte

Voici quelques fonctions communes pour le traitement du texte.

`Date currentDate()`

Retourne la date courante.

`void dateFormat(String format)`

Modifie le format de toutes les dates affichées ensuite par le modèle.

`void numberFormat(String format)`

Modifie le format de tous les nombres affichés ensuite par le modèle (non compris les nombres littéraux de type chaîne).

`void nullFormat(String format)`

Définit la chaîne utilisée à la place de `null` lors de l'affichage d'une variable dont la valeur est nulle.

`void setLocale(String language, String country)`

Modifie la locale de toutes les dates, de tous les nombres, etc., affichés ensuite par le modèle.

`boolean startsWith(String str, String prefix)`

Retourne `true` si la chaîne commence par le préfixe indiqué.

`boolean endsWith(String str, String suffix)`

Retourne `true` si la chaîne se termine par le suffixe indiqué.

`String substring(String str, int startIndex, int endIndex)`

Retourne la partie de chaîne commençant à l'indice de départ et se terminant à l'indice de fin (exclus).

`String trim(String str)`

Retourne la chaîne dont les espaces de début et de fin ont été enlevés.

```
int[] find(String str, String search)
int[] find(String str, String search, int fromIndex)
```

Retourne les indices auxquels la chaîne indiquée contient la chaîne recherchée, avec un indice de début optionnel.

```
String replace(String source, String pattern, String replacement)
String replace(String source, String pattern, String replacement,
               int fromIndex)
String replace(String source, String pattern, String replacement,
              int fromIndex, int toIndex)
```

Remplace toutes les correspondances exactes du motif dans la source par la chaîne indiquée, avec des indices de début et de fin optionnels.

Traitement du contenu

Voici quelques fonctions communes pour le traitement du contenu.

`void insertFile(String filename)`

Insère le contenu du fichier indiqué dans la page. N'insère rien si le fichier ne peut être lu. Le nom du fichier peut être absolu ou relatif. S'il est absolu, le chemin commence à partir de la racine du modèle (habituellement la racine des documents de l'application web).

`void insertURL(String url)`

Insère le contenu de l'URL dans la page. Si l'URL ne peut être lue, rien n'est inséré. L'URL peut être relative ou absolue.

`boolean fileExists(String filename)`

Retourne true si le fichier de nom indiqué existe.

`boolean URLExists(String url)`

Retourne true si l'URL indiquée existe.

Traitement de la requête et de la réponse

Il existe aussi quelques méthodes de requête/réponse fréquemment utilisées.

```
com.go.teaservlet.HttpContext.Request getRequest()
com.go.teaservlet.HttpContext.Request getRequest(String encoding)
```

Obtient un objet qui contient les informations de requêtes, avec un codage optionnel autorisant la conversion automatique des valeurs des paramètres.

`void setContentType(String contentType)`

Fixe le type de contenu de la réponse.

`void setHeader(String name, String value)`

Fixe un en-tête de réponse.

`void setStatus(int sc)`

Fixe le code d'état de la réponse.

`void sendRedirect(String url)`

Redirige la requête vers l'URL indiquée.

```
void sendError(int sc)
void sendError(int sc, String message)
```

Envoie une page d'erreur correspondant au code d'état indiqué accompagnée d'un message d'erreur optionnel.

Le modèle de l'*exemple 14-5* montre comment un modèle peut appeler les fonctions utilitaires pour afficher le pourcentage de clics sur une publicité effectués pendant une journée. La date et le pourcentage sont mis en forme suivant le style français.

Exemple 14-5. Clics français

```
<% template clicks() %>

<HTML><HEAD><TITLE>Tester les fonctions intégrées de Tea</TITLE></HEAD><BODY>

<%
// Définir la locale ; elle reste active pour toutes les sorties ultérieures
// du modèle.
setLocale("fr", "CA")  // Québec

// Spécifier la mise en forme des heures et des nombres.
dateFormat("d MMMM yyyy")
numberFormat("#0.00%")

// Des données factices.
clicks = 485.0
accesses = 7673.0
%>

<H2><% currentDate() %></H2>
Pourcentage : <% clicks/accesses %>

</BODY></HTML>
```

Le modèle affiche une sortie semblable à la suivante, la date et le pourcentage étant adaptés à une personne vivant au québec :

```
6 août 2002
Pourcentage : 6,32%
```

Écrire une application Tea

Examinons maintenant comment écrire une application Tea personnalisée. C'est facile : il suffit d'écrire une classe d'application et une classe de contexte. Toutes les classes d'application implémentent l'interface `com.go.teaservlet.Application`. Tout comme une servlet, l'interface `Application` possède des méthodes `init()` et `destroy()` pour gérer les problèmes du cycle de vie. À la place de la méthode `service()`, `Application` propose les méthodes `createContext()` et `getContextType()`. La méthode `createContext()` retourne un objet de contexte dont les méthodes sont disponibles au modèle sous forme de fonctions. La méthode `getContextType()` renvoie le `java.lang.Class` de l'objet retourné par `createContext()` afin de faciliter le contrôle de type.

Les deux exemples suivants montrent une application simple qui essaie de déterminer de différentes manières le nom de l'utilisateur. La classe `NameApp` est la classe d'application (*exemple 14-6*) ; la classe `NameContext` est l'objet de contexte créé par `NameApp` et rendu disponible aux modèles (*exemple 14-7*).

Exemple 14-6. La classe NameApp

```
import com.go.teaservlet.*;
import javax.servlet.*;

public class NameApp implements Application {

  // init(ApplicationConfig) et destroy() doivent être implémentées car elles
  // sont déclarées dans l'interface Application. Elles peuvent être vides.
  public void init(ApplicationConfig config) throws ServletException {
  }

  // La création d'un contexte apporte des fonctions disponibles aux modèles.
  public Object createContext(ApplicationRequest request,
                              ApplicationResponse response) {
    // La requête et la réponse sont souvent en paramètres même s'ils ne
    // sont pas utilisés, car ils pourront l'être plus tard.
    return new NameContext(request, response);
  }

  // Cette méthode doit être implémentée de manière à retourner la classe
  // de l'objet créé par createContext().
  public Class getContextType() {
    return NameContext.class;
  }

  public void destroy() {
  }
}
```

TeaServlet appelle la méthode `init()` une seule fois lorsque l'application est chargée pour la première fois afin de lui donner l'opportunité de s'initialiser. Il appelle `destroy()` une seule fois au moment de l'arrêt afin que l'application puisse procéder à son nettoyage. La méthode `init()` accepte un paramètre `ApplicationConfig` qui est une sous-interface de `ServletConfig` avec trois méthodes additionnelles, `getLog()`, `getName()` et `getProperties()`, qui permettent de retrouver le journal, le nom et les propriétés d'initialisation de l'application.

TeaServlet appelle la méthode `createContext()` avant le traitement d'une requête. Elle rend toutes les méthodes de l'objet de contexte retourné disponibles sous forme de fonctions au modèle Tea demandé. S'il faut un contexte différent pour chaque utilisateur, il suffit d'employer le code de suivi de session classique dans la méthode `createContext()`. La méthode `createContext()` accepte des paramètres `ApplicationRequest` et `ApplicationResponse`. Ce sont des sous-interfaces de `HttpServletRequest` et de `HttpServletResponse` avec quelques méthodes Tea supplémentaires.

Exemple 14-7. La classe NameContext

```
import com.go.teaservlet.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Exemple 14-7. La classe NameContext (suite)

```
public class NameContext {

  ApplicationRequest request;
  ApplicationResponse response;
  String name;

  public NameContext(ApplicationRequest request,
                     ApplicationResponse response) {
    this.request = request;
    this.response = response;
  }

  public String getName() {
    // Si nous avons déjà déterminer le nom de l'utilisateur, le renvoyer.
    if (name != null) {
      return name;
    }

    // Tenter de déterminer le nom de l'utilisateur.
    name = request.getRemoteUser();

    // Si le nom d'ouverture de session n'est pas disponible, essayer de
    // lire un paramètre.
    if (name == null) {
      name = request.getParameter("name");
    }

    // Si le nom n'est pas disponible dans un paramètre, essayer la session.
    if (name == null) {
      name = (String) request.getSession().getAttribute("name");
    }

    // Si le nom n'est pas dans la session, essayer un cookie.
    if (name == null) {
      Cookie[] cookies = request.getCookies();
      for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("name")) {
          name = cookies[i].getValue();
        }
      }
    }

    // Si le nom n'est pas dans un cookie, abandonner.
    return name;
  }
}
```

La méthode getName() tente de déterminer le nom de l'utilisateur en recherchant dans les certificats du client, dans la liste des paramètres, dans les données de session et dans les cookies de la requêtes. Elle retourne null si elle ne trouve aucune information de nom. Les modèles accèdent au nom en appelant la fonction getName(). Notez que le contexte détermine le nom du client dans getName()et non dans le constructeur. Il s'agit d'une astuce pour améliorer les performances. Si la logique se trouvait dans le constructeur, elle serait exécutée pendant le traitement de chaque requête, que le modèle appelle ou non la méthode getName().

Pour que l'application NameApp soit disponible, il est nécessaire d'ajouter les lignes suivantes au fichier *TeaServlet.properties* :

```
"NameApp" {
    class = NameApp
}
```

Vous pouvez vérifier que l'application a été correctement chargée en parcourant le lien Applications de l'application d'administration de TeaServlet.[3] Avec cette application chargée, tout modèle peut afficher le nom du client à l'aide de la fonction getName() :

```
<% template names() %>

<%
setContentType("text/plain")
nullFormat("Inconnu")
%>

Votre nom est <% getName() %>.
```

Une application outil

Déterminer le nom d'un utilisateur peut difficilement être considéré comme une application réelle. Continuons donc par quelque chose qui s'en approche plus. Nous allons créer une application qui affiche une liste des différents outils de création de contenu disponibles (une application dont Servlets.com a besoin). Les informations sur les outils proviennent d'un fichier XML, même si elles pourraient venir d'une base de données. L'*exemple 14-8* montre la classe ToolApp.

3. Si les classes de l'application ne sont pas trouvées alors qu'elles devraient l'être, le problème peut venir du chargeur de classes. Il peut y avoir des problèmes si *TeaServlet.jar* et les classes de l'application sont chargées par des chargeurs différents. Vous devez, soit placer les deux dans le chemin de classe du systèmes où ils seront trouvés par le chargeur de classes principal, soit les placer sous le répertoire *WEB-INF* où ils seront trouvés par le chargeur de classes de l'application web. (Mettez *TeaServlet.jar* dans le répertoire *WEB-INF/lib* ; mettez les classes d'application dans le répertoire *WEB-INF/classes*.) Rappelez-vous aussi que, du fait des problèmes du chargeur de classes, les classes support ne sont pas forcément rechargées même après avoir cliqué sur Reload All dans la page d'administration de TeaServlet.

Exemple 14-8. Le cerveau de l'application outil

```
import com.go.teaservlet.*;
import com.go.trove.log.*;
import java.sql.Timestamp;
import java.util.*;
import javax.servlet.*;

public class ToolApp implements Application {

  private Log log;
  private Tool[] tools;

  public void init(ApplicationConfig config) throws ServletException {
    // Conserver un journal des événements spécifique à cette application.
    log = config.getLog();

    // Charger les données sur les outils dans l'initialisation pour plus
    // de simplicité.
    String toolsFile = config.getInitParameter("toolsFile");
    if (toolsFile == null) {
      throw new ServletException(
        "Un fichier de données sur les outils doit être spécifié dans " +
        "le paramètre d'initialisation toolsFile");
    }
    log.debug("Chargement des outils à partir de " + toolsFile);
    try {
      tools = Tool.loadTools(toolsFile);
      if (tools.length == 0) {
        log.warn("Aucun outil trouvé dans " + toolsFile);
      }
    }
    catch (Exception e) {
      log.error(e);
      throw new ServletException(e);
    }
  }

  public Object createContext(ApplicationRequest request,
                              ApplicationResponse response) {
    return new ToolContext(request, response, this);
  }

  public Class getContextType() {
    return ToolContext.class;
  }

  public void destroy() {
  }
```

Exemple 14-8. Le cerveau de l'application outil (suite)

```
  public Tool[] getTools() {
    // Normalement l'"application" devrait maintenir ou avoir un accès à une
    // connexion de base de données existante. Ici, pour plus de simplicité,
    // nous utilisons XML.
    return tools;
  }

  public Tool[] getTools(String state) {
    // Retourner les outils dans un état donné (soumis, actifs, rejetés ou
    // morts).
    List list = new LinkedList();
    for (int i = 0; i < tools.length; i++) {
      if (tools[i].getStateFlag().equalsIgnoreCase(state)) {
        list.add(tools[i]);
      }
    }
    return (Tool[]) list.toArray(new Tool[0]);
  }
}
```

La méthode init() de ToolApp utilise l'ApplicationConfig passé pour retrouver et sauvegarder une référence au journal. Puis elle utilise config pour obtenir la valeur du paramètre d'initialisation toolsFile. La valeur de ce paramètre doit être spécifiée dans le fichier *TeaServlet.properties* :

```
  "ToolApp" {
      class = ToolApp
      init {
          toolsFile = /tomcat/webapps/teatime/WEB-INF/tools.xml
      }
  }
```

La méthode init() appelle ensuite loadTools() pour charger les informations sur les outils dans un tableau à partir du fichier XML. Le fichier XML doit ressembler à celui montré à l'*exemple 14-9*.

Exemple 14-9. Exemple de données

```
<?xml version="1.0" encoding="ISO-8859_1"?>

<tools>
  <tool id="1">
    <name>JavaServer Pages</name>
    <homeURL>http://java.sun.com/products/jsp</homeURL>
    <comments>
       Les JSP (JavaServer Pages) sont une technologie créée par Sun
       Microsystems et intimement liée aux servlets. Comme pour les
       servlets, Sun a publié une spécification des JSP et des fournisseurs
       tiers mettent en concurrence leur implémentation de cette norme. En
       étant publiées par Sun, les JSP sont dans une position très
       privilégiée et si les JSP avaient résolu un nombre suffisamment
```

Exemple 14-9. Exemple de données (suite)

```
      important de problèmes des utilisateurs, elles auraient certainement
      gagné le marché avant que d'autres solutions viables existent. En
      fait, un nombre surprenant d'utilisateurs sont désenchantés par les
      JSP et les autres approches gagnent en popularité.
    </comments>
    <stateFlag>LIVE</stateFlag>
    <createdTime>1998-03-17 00:00:00.000</createdTime>
    <modifiedTime>1999-12-16 00:00:00.000</modifiedTime>
  </tool>
  <tool id="2">
    <name>Tea</name>
    <homeURL>http://opensource.go.com</homeURL>
    <comments>
      Tea est un nouveau produit en open source du Walt Disney
      Internet Group (précédemment GO.com) créé en interne ces dernières
      années afin de résoudre les besoins énormes de la production des
      sites web tels que ESPN.com. Tea est similaire aux JSP tout en
      évitant beaucoup de leurs problèmes. Il possède déjà un grand
      nombre d'outils.
    </comments>
    <stateFlag>LIVE</stateFlag>
    <createdTime>2000-07-12 00:00:00.000</createdTime>
    <modifiedTime>2000-07-12 00:00:00.000</modifiedTime>
  </tool>
  <tool id="3">
    <name>WebMacro</name>
    <homeURL>http://jakarta.apache.org</homeURL>
    <comments>
      WebMacro est un moteur de modèle créé par Semiotek, Inc., pour le
      projet Shimari et aujourd'hui inclus dans Apache Jakarta Project.
      WebMacro a été utilisé pour des sites commerciaux comme
      AltaVista.com, il a été intégré à des frameworks en open source
      comme Turbine et Melati, et il a été utilisé dans des gros projets
      en open source comme JetSpeed.
    </comments>
    <stateFlag>LIVE</stateFlag>
    <createdTime>1998-11-19 00:00:00.000</createdTime>
    <modifiedTime>2000-08-31 00:00:00.000</modifiedTime>
  </tool>
  <tool id="4">
    <name>Element Construction Set</name>
    <homeURL>http://java.apache.org/ecs</homeURL>
    <comments>
      Le paquetage ECS (Element Construction Set) du Apache Jakarta
      Project est un ensemble de classes modélisées d'après le produit
      htmlKona de WebLogic (maintenant BEA Systems). ECS a beaucoup de
      limitations, mais il résout un certain nombre de problèmes.
    </comments>
    <stateFlag>LIVE</stateFlag>
```

Exemple 14-9. Exemple de données (suite)

```
    <createdTime>1999-03-31 00:00:00.000</createdTime>
    <modifiedTime>2000-06-16 00:00:00.000</modifiedTime>
  </tool>
  <tool id="5">
    <name>XMLC</name>
    <homeURL>http://xmlc.enhydra.org</homeURL>
    <comments>
      XMLC utilise XML pour exploiter toute la puissance d'ECS sans en
      avoir les limitations. Il a été créé par Lutris et il peut être
      utilisé comme un composant séparé.
    </comments>
    <stateFlag>LIVE</stateFlag>
    <createdTime>1998-10-11 00:00:00.000</createdTime>
    <modifiedTime>2000-03-09 00:00:00.000</modifiedTime>
  </tool>
  <!-- etc -->
</tools>
```

L'application lit ce fichier à l'aide de l'API JDOM, une API en *open source* pour lire, écrire et manipuler du XML avec Java. Si nous voulions lire les résultats à partir d'une base de données, nous pourrions simplement remplacer le `SAXBuilder` de JDOM (qui construit un `Document` à partir d'un fichier ou d'un flot à l'aide d'un analyseur syntaxique SAX) par un `ResultSetBuilder` JDOM (qui construit un `Document` à partir d'un `java.sql.ResultSet`, une classe offerte à JDOM).[4] Au moment de l'écriture de ce livre, JDOM est encore en développement. Cet exemple utilise JDOM Beta 5. Des modifications mineures peuvent être nécessaires pour fonctionner avec la version finale de JDOM.

La méthode `createContext()` construit une nouvelle instance de `ToolContext`, dont le code est montré à l'*exemple 14-10*.

Exemple 14-10. Le contexte contenant toutes les fonctions disponibles

```
import com.go.teaservlet.*;

public class ToolContext {

  ApplicationRequest request;
  ApplicationResponse response;
  ToolApp app;

  public ToolContext(ApplicationRequest request, ApplicationResponse response,
                     ToolApp app) {
    this.request = request;
    this.response = response;
```

4. Jason est l'un des cocréateurs de JDOM, avec Brett McLaughlin. Pour plus d'informations sur JDOM, allez à *http://jdom.org*.

Exemple 14-10. Le contexte contenant toutes les fonctions disponibles (suite)

```
    this.app = app;
  }

  public Tool[] getTools() {
    return app.getTools();
  }

  public Tool[] getTools(String state) {
    return app.getTools(state);
  }
}
```

Les méthodes publiques du contexte sont les fonctions rendues disponibles aux modèles Tea. Ici, les méthodes appellent l'application. Ce style de programmation est considéré comme bon car les contextes doivent être aussi légers que possibles ; l'état doit se trouver dans l'application. La classe `Tool` agit elle-même comme le conteneur des informations de `Tool`. L'*exemple 14-11* montre ce code.

Exemple 14-11. Une classe pour charger et contenir les données sur les outils

```
import java.io.*;
import java.sql.*;
import java.util.*;
import org.jdom.*;
import org.jdom.input.*;

public class Tool {
  // Données sur l'outil.
  public int id;
  public String name;
  public String homeURL;
  public String comments;
  public String stateFlag;
  public Timestamp createdTime;
  public Timestamp modifiedTime;

  // Tea ne peut accéder qu'aux propriétés du bean, les méthodes d'accès
  // sont donc nécessaires.
  public int getId() { return id; }
  public String getName() { return name; }
  public String getHomeURL() { return homeURL; }
  public String getComments() { return comments; }
  public String getStateFlag() { return stateFlag; }
  public Timestamp getCreatedTime() { return createdTime; }
  public Timestamp getModifiedTime() { return modifiedTime; }

  public int getCreatedAgeInDays() {
    return (int) ((System.currentTimeMillis() - createdTime.getTime()) /
            (24 * 60 * 60 * 1000));  // Millisecondes dans une journée.
```

Exemple 14-11. Une classe pour charger et contenir les données sur les outils (suite)

```
  }

  public int getModifiedAgeInDays() {
    return (int) ((System.currentTimeMillis() - modifiedTime.getTime()) /
            (24 * 60 * 60 * 1000));  // Millisecondes dans une journée.
  }

  // Idéalement, nous devrions utiliser des méthodes comme celles-ci, mais
  // Tea n'accepte que des accès aux propriétés d'un objet. Celles-ci ne
  // sont pas visibles.
  public boolean isNewWithin(int days) {
    return getCreatedAgeInDays() < days;
  }

  public boolean isUpdatedWithin(int days) {
    return getModifiedAgeInDays() < days;
  }


  public static Tool[] loadTools(String toolsFile) throws Exception {
    // Lire les données sur les outils à partir du fichier XML contenant
    // des éléments <tool>. Utiliser l'API JDOM pour que les choses
    // restent simples (http://jdom.org).
    List toolObjects = new LinkedList();

    SAXBuilder builder = new SAXBuilder();
    Document document = builder.build(new File(toolsFile));
    Element root = document.getRootElement();
    List toolElements = root.getChildren("tool");
    Iterator i = toolElements.iterator();
    while (i.hasNext()) {
      Element tool = (Element) i.next();
      Tool t = new Tool();
      t.id = tool.getAttribute("id").getIntValue();
      t.name = tool.getChild("name").getTextTrim();
      t.homeURL = tool.getChild("homeURL").getTextTrim();
      t.comments = tool.getChild("comments").getTextTrim();
      t.stateFlag = tool.getChild("stateFlag").getTextTrim();
      t.createdTime = Timestamp.valueOf(
                        tool.getChild("createdTime").getTextTrim());
      t.modifiedTime = Timestamp.valueOf(
                         tool.getChild("modifiedTime").getTextTrim());
      toolObjects.add(t);
    }

    return (Tool[]) toolObjects.toArray(new Tool[0]);
  }
}
```

Notez que même si les variables contenant l'état de la classe sont publiques, elles ne sont pas visibles à partir des modèles Tea. Les modèles Tea ont uniquement accès aux fonctions déclarées par les contextes installés et aux propriétés des objets retournés par ces fonctions. C'est pourquoi, la classe `Tool` offre diverses méthodes d'accès. `Tool` a aussi deux méthodes utilitaires pour envoyer le nombre de jours écoulés depuis la création et la modification de l'enregistrement. On pourrait être tenté d'utiliser les méthodes `isNewWithin(int days)` ou `isUpdatedWithin(int days)`. Cependant, le modèle ne voit pas ces méthodes car elles n'apparaissent pas comme une propriété de bean.

L'*exemple 14-12* montre un modèle frontal simple pour cette application. Il s'appelle `toolview1` et il accepte un paramètre optionnel `state` afin de limiter la liste des outils dans un état spécifique. Une copie d'écran de sa sortie est montrée à la *figure 14-3*.

Exemple 14-12. Une vue simple des outils

```
<% template toolview1(String state) %>

<%
  if (state == null) {
    tools = getTools("live");
  }
  else {
    tools = getTools(state)
  }
%>

<% foreach (tool in tools) { %>

  <HR SIZE=2 ALIGN=LEFT>

  <H3>
  <% tool.name %>
  <%
  if (tool.createdAgeInDays < 45) {
    '<FONT COLOR=#FF0000><B> (New!) </B></FONT>'
  }
  else if (tool.modifiedAgeInDays < 45) {
    '<FONT COLOR=#FF0000><B> (Updated!) </B></FONT>'
  }
  %>
  </H3>
  <A HREF="<% tool.homeURL %>"><% tool.homeURL %></A><BR>

  <% tool.comments %>

<% } %>
```

Comme vous pouvez le voir, la sortie manque de décoration (en-tête, barre de menu, bas de page) comme le voudrait une page professionnelle. Nous pouvons l'ajouter en appelant d'autres modèles, comme le montre à l'*exemple 14-13* la version revue de `toolview2`.

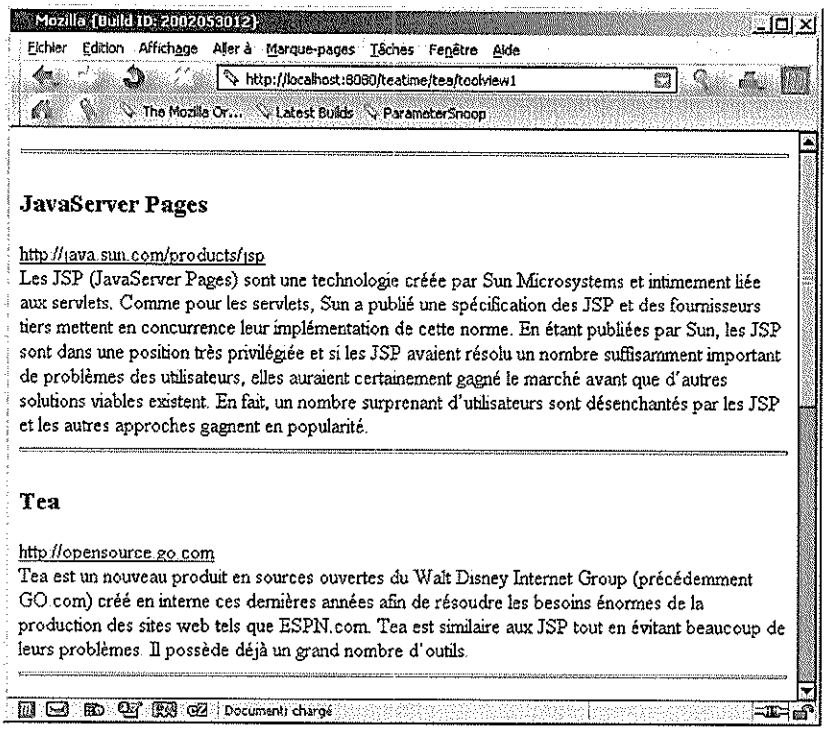


Figure 14-3. Une liste des outils sans contenu d'enrobage

Exemple 14-13. Une vue plus complète des outils

```
<% template toolview2(String state) %>

<%
    title = "Liste des outils"
    deck = "Une liste des outils de création de contenu"
    desc = "Sans les outils, les gens ne sont rien de plus que des " &
           "animaux. Et plutôt faibles, en plus. Voici une liste " &
           "d'outils de création de contenu basé sur les servlets que " &
           "vous pouvez utiliser afin de ne plus être une mauviette."
%>

<% call header(title, deck, desc) %>

<%
  if (state == null) {
    tools = getTools("live");  // Le ; est autorisé mais non requis.
  }
  else {
    tools = getTools(state)
  }
%>
```

Exemple 14-13. Une vue plus complète des outils (suite)

```
<% foreach (tool in tools) { %>

  <HR SIZE=2 ALIGN=LEFT>

  <H3>
  <% tool.name %>
  <%
  if (tool.createdAgeInDays < 45) {
    '<FONT COLOR=#FF0000><B> (New!) </B></FONT>'
  }
  else if (tool.modifiedAgeInDays < 45) {
    '<FONT COLOR=#FF0000><B> (Updated!) </B></FONT>'
  }
  %>
  </H3>
  <A HREF="<% tool.homeURL %>"><% tool.homeURL %></A><BR>

  <% tool.comments %>

<% } %>

<% call footer() %>
```

Ce nouveau modèle définit des variables pour le titre, le sujet et la description. Ensuite il appelle un modèle d'en-tête en lui passant ces valeurs. Ce modèle génère le contenu de l'en-tête et de la barre de menu, puis il laisse le modèle `toolview2` ajouter le contenu fondamental. À la fin de la page, un modèle ajoute un bas de page. Des exemples de modèles d'en-tête et de base de page sont montrés à l'*exemple 14-14* et à l'*exemple 14-15*. La sortie améliorée est visible à la *figure 14-4*.

Exemple 14-14. Le fichier d'en-tête

```
<% template header(String title, String deck, String desc) %>

<HTML><HEAD><TITLE><% title %></TITLE></HEAD>

<BODY BGCOLOR="#FFFFFF" BACKGROUND="/images/background.gif"
      LINK="#003333" ALINK="#669999" VLINK="#333333">

<IMG SRC="/images/banner.gif" WIDTH=600 HEIGHT=87 BORDER=0><BR>

<TABLE>
<TR>
<TD WIDTH=125 VALIGN=TOP>
 <BR><BR><BR>
 <FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#FF0000">
 <A HREF="/index.html">Accueil</A><BR>
 <A HREF="/hosting.html">Hébergement</A><BR>
 <A HREF="/engines.html">Moteurs</A><BR>
```

Exemple 14-14. Le fichier d'en-tête (suite)

```
 </FONT>
</TD>

<TD WIDTH=475>

  <TABLE CELLPADDING=5><TR><TD WIDTH=600 BGCOLOR="#006699" VALIGN=TOP>
  <B><FONT FACE="Arial,Helvetica" SIZE="+2">
  <% title %>
  </FONT></B>
  </TD></TR></TABLE>

  <B><FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#003366">
  <% deck %>
  </FONT></B><P>

  <P>

  <FONT FACE="Arial,Helvetica">

  <% desc %>
```

Exemple 14-15. Le fichier de bas de page

```
<% template footer() %>

 </FONT>

</TD>
</TR>

<TR>
<TD></TD>
<TD WIDTH=475 ALIGN=CENTER COLSPAN=3>
<HR>
<FONT FACE="Arial,Helvetica">
<A HREF="/index.html">Accueil</A>  
<A HREF="/hosting.html">Hébergement</A>  
<A HREF="/engines.html">Moteurs</A>  <P>
</FONT>

<TABLE WIDTH=100%>
<TR>
<TD WIDTH=260 ALIGN=LEFT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
<A HREF="/copyright.html">Copyright</A> &copy; 2000 Jason Hunter<BR>
Tous droits réservés.</TD>
<TD WIDTH=5></FONT></TD>
<TD WIDTH=230 ALIGN=RIGHT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
```

Exemple 14-15. Le fichier de bas de page (suite)

```
Contact : <A HREF="mailto:webmaster@servlets.com">webmaster@servlets.com</A>
</FONT></TD>
</TR>
</TABLE>

</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Figure 14-4. Une liste des outils avec du contenu d'enrobage

En suivant cette conception, les modifications de l'en-tête, de la barre de menu ou du bas de page sont isolées dans des fichiers séparées. Il est aussi possible d'utiliser un seul fichier pour la mise en forme de la page complète et d'ajouter le contenu fondamental à l'aide de la commande <% ... %>. Cette approche apporte plus de puissance et permet à la partie fondamentale de ne rien savoir de sa mise en forme externe, même s'il faut un peu plus de temps pour s'y habituer. Assurez-vous de lire la documentation de TeaServlet pour plus d'informations.

Pour finir

L'une des fonctionnalités les plus utiles de Tea est difficile à montrer dans un livre : son environnement graphique de création appelé Kettle (voir la *figure 14-5*). En plus des fonctionnalités traditionnelles d'un IDE, Kettle fournit un retour en temps réel sur le processus de création de modèle, soulignant, pendant la saisie, les erreurs de syntaxe en vert, les erreurs sémantiques en rouge et les erreurs de compilation en bleu. Une fonctionnalité similaire dans des outils supportant d'autres mécanismes de création de contenu serait appréciée mais quasiment impossible à mettre en œuvre. Kettle offre aussi un examen contextuel des propriétés afin d'afficher immédiatement au créateur de modèle les propriétés disponibles sur un objet donné. Kettle est disponible gratuitement mais ne fonctionne que sur Windows et n'est pas en *open source*. Il existe des projets d'en réaliser une version en *open source* et basée sur Swing.

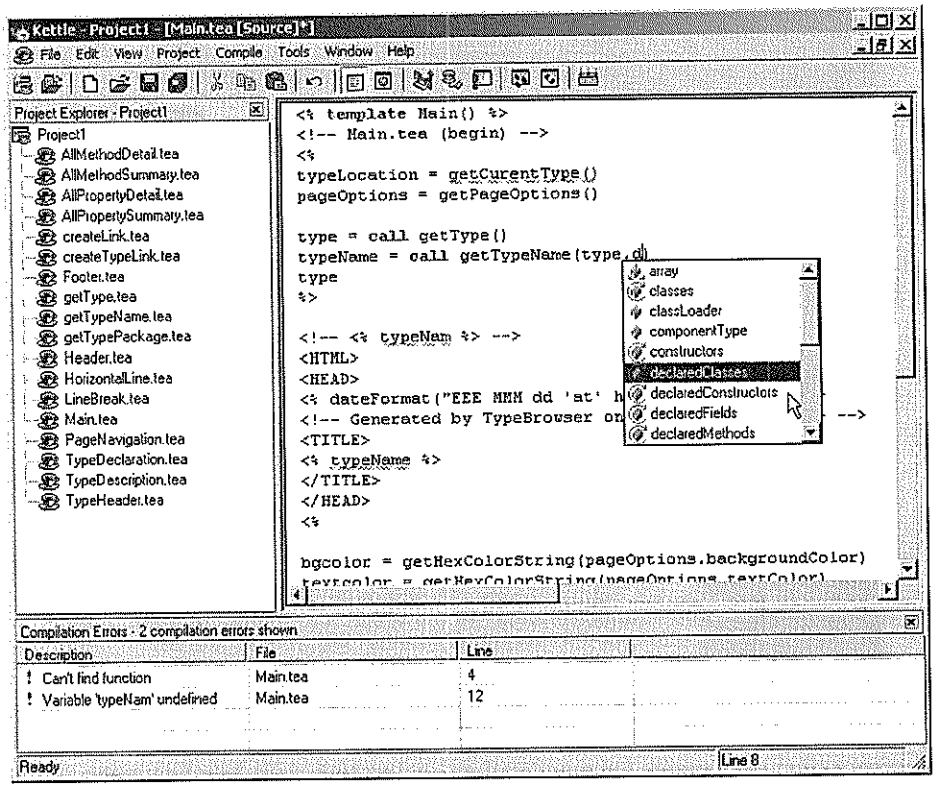


Figure 14-5. Il vaut mieux faire le thé (Tea) dans une bouilloire (Kettle)

Au fur et à mesure que vous explorerez Tea et TeaServlet, vous pourrez constater qu'il manque le support de telle ou telle fonctionnalité spéciale car ses créateurs (WDIG) n'en avaient pas besoin. Les sources étant disponibles, d'autres utilisateurs vont ajouter leurs besoins et leurs idées, le produit va se développer. Bien que le langage Tea soit mature, TeaServlet est encore assez récent et dispose d'un grand potentiel.

15

WebMacro

Dans la même famille que Tea, mais avec une approche plus orientée servlets, il existe un environnement de développement appelé WebMacro, disponible sur le site *http://www.webmacro.org*. WebMacro a été créé par Justin Wells de Semiotek dans le cadre d'un contrat de création du site de la société et il est devenu en *open source* afin de l'améliorer et d'augmenter la visibilité de la société. Des discussions sont en cours pour l'intégration de WebMacro dans l'Apache Jakarta Project à *http://jakarta.apache.org* afin d'élargir le développement communautaire. Il existe aujourd'hui un clone de WebMacro appelé Velocity et développé au sein de Jakarta.

WebMacro est souvent désigné comme un *moteur de templates*, le terme générique pour des systèmes qui remplacent du texte dans des modèles de pages web. Il existe d'autres moteur de templates en *open source*, en particulier FreeMarker (*http://freemarker.sourceforge.net*) ; cependant, nous examinons ici WebMacro car au moment de l'écriture de ce livre, il dispose de la plus large audience, a été utilisé sur de grands sites commerciaux comme AltaVista, a été intégré dans des environnements en *open source* comme Turbine (*http://java.apache.org/turbine*) et Melati (*http://www.melati.org*) et il a été utilisé comme base pour des projets en *open source* importants comme JetSpeed (*http://java.apache.org/jetspeed*). Comme Tea, la licence de WebMacro est du type Apache afin qu'il puisse être employé sans problèmes en association à des applications commerciales.

Cette section décrit WebMacro 0.94, la préversion disponible au mois d'octobre 2000.[1] Plusieurs points doivent être modifiés et le seront. Si les exemples de code donnés ici ne fonctionnent pas avec la version finale, lisez la documentation fournie par WebMacro pour déterminer ce qu'il faut changer. WebMacro nécessite l'API Servlet 2.0, ou ultérieure, et le JDK 1.1.7, ou ultérieur.

1. *N.d.T.* : au moment de la traduction de ce livre, la version courante de WebMacro est la version 1.0.1. Elle nécessite le JDK 1.2, ou ultérieur.

L'environnement WebMacro

L'environnement WebMacro fonctionne de la façon suivante : une servlet reçoit une requête d'un client, elle applique sa logique métier nécessaire au traitement de la requête (certainement en utilisant des classes support ou des composants comme des EJB) et elle crée un objet de contexte rempli « d'objets de réponse » contenant les résultats issus de la logique métier et devant être affichés au client. La servlet sélectionne ensuite un fichier modèle (peut-être selon les résultats) et passe les objets de réponse au travers du fichier modèle pour générer le contenu destiné au client.

Un programmeur Java crée la servlet et la logique métier en code Java pur, et un concepteur de modèles crée le fichier modèle pour l'affichage. Un modèle est constitué de contenu HTML ou XML rempli à l'aide d'une syntaxe de substitution simple et de petites possibilités de script. La syntaxe est compatible avec les éditeurs de HTML et de XML, évitant explicitement l'emploi des symboles < et > afin de ne pas perturber les analyseurs syntaxiques HTML et XML. Le programmeur fournit au concepteur de modèles une liste des variables présentes, ainsi qu'une liste des propriétés et méthodes de ces variables.

Le modèle présente quelques similitudes avec l'architecture de TeaServlet, la différence étant que WebMacro utilise par défaut un modèle d'envoi plus orienté servlet, ne fournissant aucune isolation des modèles, et il utilise une syntaxe plus simple pour les scripts. Nous pensons que WebMacro est plus adapté que Tea pour les applications web hautement fonctionnelles ayant un point de terminaison, et que Tea est plus adapté que WebMacro pour les sites qui ajoutent continuellement de nouveaux contenus de modèles et nécessitant l'isolation des modèles et moins d'implications du programmeur.

Bonjour avec WebMacro

L'*exemple 15-1* montre une servlet simple utilisant le système de modèles de WebMacro. L'*exemple 15-2* montre le fichier modèle *hello.wm*. Ensemble, ils affichent la date et l'heure courantes. Nous expliquerons où placer ces fichiers à la prochaine section.

Exemple 15-1. Une servlet simple pour piloter le modèle Hello

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.webmacro.*;
import org.webmacro.servlet.*;

public class WMHello extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    FastWriter out = new FastWriter(res.getOutputStream(),
                                    res.getCharacterEncoding());

    try {
      WebMacro wm = new WM(); // Ou WM("/path/to/config/file").
```

Exemple 15-1. Une servlet simple pour piloter le modèle Hello (suite)

```
      Context c = wm.getWebContext(req, res);
      c.put("date", new Date());
      Template tmpl = wm.getTemplate("hello.wm");
      tmpl.write(out, c);
      out.flush();
    } catch (WebMacroException e) {
      throw new ServletException(e);
    }
  }
}
```

Exemple 15-2. Un modèle Hello simple

```
## hello.wm

#set $Response.ContentType = "text/html"

<HTML><HEAD><TITLE>Test de WebMacro</TITLE></HEAD><BODY>

Bonjour ! <P>

Nous sommes actuellement le $date. <BR>
(Pour vous les accros, cela fait $date.Time.) <P>

</BODY></HTML>
```

Commençons par la servlet. Il s'agit d'une servlet normale, étendant `HttpServlet` et implémentant la méthode `doGet()`. Vous noterez tout de suite que sa logique est différente. Elle retrouve un `FastWriter` à la place d'un `PrintWriter` normal. Cela autorise des optimisations spéciales évitant le coût de la conversion Unicode pour le contenu statique. Il fonctionne comme un écrivain ordinaire amélioré de manière à permettre à WebMacro de lui passer des octets préencodés et d'appeler `setAsciiHack(true)` afin d'augmenter la vitesse lorsque les données sorties sont du type Latin-1 ou US-ASCII. Assurez-vous simplement que (tout au moins pour le moment) vous appelez bien la méthode `flush()` de l'écrivain pour envoyer au client le contenu du tampon.

Le code dans le bloc `try` crée un nouvel objet `WebMacro` représentant le lien principal avec le système de WebMacro. `WebMacro` étant une interface, la servlet construit en fait une instance de la classe concrète `WM` qui implémente l'interface `WebMacro`. La servlet appelle ensuite `wm.getWebContext(req, res)` pour retrouver un `WebContext` dans lequel seront placés les objets de réponse passés au modèle. Le contexte fonctionne comme un `Hashtable`. Il offre une méthode `put(Object, Object)` que la servlet utilise pour placer un objet `Date` dans le contexte sous le nom `date`. Cet objet sera ensuite disponible au modèle s'exécutant dans ce contexte. Pour les utilisateurs avancés, il est intéressant de noter que la classe `WebContext` étend `Context` et que la servlet peut retrouver un `Context` non web en appelant `getContext()`. C'est pratique dans certains cas, comme une utilisation hors-ligne ou non-servlet.

Pour retrouver un modèle, la servlet appelle `wm.getTemplate("hello.wm")`. La méthode accepte le nom du modèle incluant l'extension de fichier. Un modèle peut utiliser toute extension de fichier, mais la plupart des gens choisissent *.wm* par convention.

Avec le modèle en mains, la servlet peut appeler la méthode write() et passer en paramètre le FastWriter sur lequel écrire et le WebContext rempli d'objets de réponse. La sortie pourrait être aussi se trouver sous forme d'un String en utilisant la méthode String evaluate(Context context) de Template.

Si une méthode lance une WebMacroException, la servlet est écrite de manière à relancer l'exception enveloppée dans une ServletException. Presque toutes les méthodes WebMacro peuvent lancer une WebMacroException ou une de ses sous-classes. Par exemple, le constructeur WM() peut lancer une InitException si un fichier de configuration requis n'est pas trouvé. La méthode getTemplate() peut lancer une NotFoundException si le modèle ne peut être trouvé. La méthode write() peut lancer une ContextException si le modèle a besoin de données qui ne se trouvent pas dans le contexte et elle peut lancer une IOException s'il y a eu un problème d'écriture sur le client. Toutes les exceptions définies par WebMacro étendent WebMacroException, une exception contrôlée, et, en général, l'exception générique est interceptée et traitée à la fin de la méthode doGet().

Une autre approche consiste à écrire un servlet de type WebMacro en étendant la classe mère org.webmacro.servlet.WMServlet, comme le montre l'*exemple 15-3*.

Exemple 15-3. Une autre approche à la conception d'une servlet WebMacro

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.webmacro.*;
import org.webmacro.servlet.*;

public class WMServletHello extends WMServlet {

  public Template handle(WebContext context) throws HandlerException {
    try {
      context.put("date", new Date());
      return getTemplate("hello.wm");
    }
    catch (NotFoundException e) {
      throw new HandlerException(e.getMessage());
    }
  }
}
```

Dans ce schéma, une servlet étend WMServlet et implémente une seule méthode handle(). La classe mère crée automatiquement l'objet WebMacro et passe le WebContext dans un paramètre. La servlet doit seulement effectuer sa logique métier, remplir le WebContext avec les objets de réponse appropriés et retourner le modèle servant à la création de la page (ou null si la servlet a créé la page elle-même). Il est aussi possible d'implémenter dans la servlet des méthodes start() et stop() qui remplacent les méthodes init() et destroy() habituelles. Si les objets HttpServletRequest et HttpServletResponse sont nécessaires, ils peuvent être pris dans le WebContext à l'aide des méthodes getRequest() et getResponse().

L'emploi de la classe mère `WMServlet` peut être plus pratique que la gestion manuelle des objets WebMacro. La contrepartie est une souplesse moindre. Vous devez prendre l'approche manuelle si vous devez placer deux modèles indépendants dans une page, passer la sortie au travers d'un filtre comme un `GZIPOutputStream`, effectuer une génération hors-ligne, différencier les requêtes GET et POST et si la servlet doit étendre une autre classe mère personnalisée.

Examinons maintenant le modèle. Il ressemble à une page HTML normale à l'exception d'une petite balise. Dans le corps de la page, nous voyons que l'objet `Date`, ajouté précédemment dans le contexte sous le nom `date`, a été inclus dans la page à l'aide de la syntaxe `$date`. Nous voyons aussi sur la ligne suivante que la propriété `time` de l'objet `Date` (qui contient l'heure courante exprimée millisecondes) a été incluse à l'aide de la syntaxe `$date.Time`. Il s'agit de la syntaxe de substitution simple utilisée par WebMacro : *`$nom_variable`* affiche la valeur de la variable (convertie en `String` si nécessaire) et *`$nom_variable.Propriété`* affiche la valeur de la propriété. Les propriétés de propriétés sont accessibles à l'aide de la syntaxe *`$nom_variable.Propriété.Sous-propriété`* et, en fait, il est possible d'accéder à bien plus que les propriétés d'un bean grâce à l'introspection avancée effectuée par WebMacro, comme nous le verrons plus loin. Pour le moment, souvenez-vous simplement que lors des accès aux propriétés d'un bean, leurs noms doivent être mis en majuscule.

Si vous regardez au début du fichier du modèle, vous verrez un commentaire donnant le nom du fichier. En WebMacro, les commentaires commencent par `##` et vont jusqu'à la fin de la ligne. Juste en dessous du commentaire, se trouve une commande `#set`, ce que WebMacro appelle une *directive*. Elle définit la propriété `ContentType` de l'objet `Response` à la valeur `"text/html"`, équivalent à l'appel de `response.setContentType("text/html")`. La variable `$Response` représente la réponse de la servlet et, avec `$Request`, elle est implicitement disponible dans tous les objets `WebContext`. WebMacro dispose d'un ensemble de directives utiles pour la manipulation de variables, l'inclusion de fichiers, la logique conditionnelle et les boucles. Nous les verrons en détail plus loin.

Installer WebMacro

Pour exécuter une servlet et un modèle WebMacro, il faut passer par une petite installation. Tout d'abord, téléchargez la distribution à partir de *http://www.webmacro.org* et extrayez-la. Incluez *webmacro.jar* et *collections.jar* dans le chemin de classes de votre serveur.[2] Localisez ensuite le fichier *WebMacro.properties* qui se trouve dans la distribution et copiez-le dans un répertoire du chemin de classes de votre serveur. Notez que les classes de *webmacro.jar* se reposant sur leur chargeur de classes pour localiser le fichier, il doit se trouver dans le chemin de classes du chargeur de classes ayant chargé *webmacro.jar*. Pour être certain que tout est correct, placez tout cela, soit dans le chemin de clas-

2. L'archive *collections.jar* contient les classes Java Collections (introduites dans le JDK 1.2) de manière à pouvoir utiliser le JDK 1.1. Ce fichier JAR n'est pas nécessaire si vous utilisez la version de WebMacro compilée pour le JDK 1.2 ; cependant, au moment de l'écriture de ce livre, la version par défaut contenue dans l'archive *webmacro.jar* a été construite pour le JDK 1.1.7 et ce fichier JAR est donc nécessaire même avec le JDK 1.2 ou ultérieur.

ses du système où ils seront trouvés par le chargeur de classes principal, soit sous le répertoire *WEB-INF* où ils seront trouvés par le chargeur de classes de l'application web. (Placez *webmacro.jar* dans le répertoire *WEB-INF/lib* et *WebMacro.properties* dans le répertoire *WEB-INF/classes*.)

Le fichier *WebMacro.properties* par défaut de la distribution contient une longue liste d'options de configuration, la première partie étant montrée à l'*exemple 15-4*. En général, tous les points de ce fichier y sont documentés. Pour le moment, il vous reste simplement à configurer le paramètre TemplatePath qui indique le(s) répertoire(s) où sont stockés vos modèles.

Exemple 15-4. Un fichier WebMacro.properties standard

```
# NOTE AUX UTILISATEURS DE NT
#
# Le caractère \ est le caractère d'échappement dans un fichier de propriétés
# Java. Dans ce fichier, vous devez, soit le doubler (\\), soit utiliser le
# séparateur de fichiers de type Unix (/). Les deux doivent fonctionner. Quand
# vous définissez TemplatePath, assurez-vous aussi d'employer le séparateur de
# chemin NT (;) en place du séparateur Unix (:).

############################################################
#
# CONFIGURATION DE BASE :
#

# Vous devez au moins définir TemplatePath ! Il contient une liste de
# répertoires dans lesquels vos modèles seront recherchés, si vous donnez
# un chemin relatif. C'est une liste de répertoires séparés par un : (sur Unix)
# ou un ; (sur NT).

TemplatePath = /tomcat/webapps/webmacro/WEB-INF/templates;/local/webmacro/
templates

# Pour plus d'efficacité, WebMacro compile et place dans un cache les modèles.
# Pendant le développement, vous pouvez désactiver cette option en mettant la
# valeur suivante à 0 afin que les modifications apportées à votre modèle
# soient vues immédiatement sur le site web. Pour les systèmes en production,
# il s'agit du nombre de millisecondes d'inactivité pendant lequel le modèle
# dans le cache sera conservé, par exemple 600000 pour dix minutes.

TemplateExpireTime = 0
# TemplateExpireTime == 600000

# LogLevel peut être : ALL, DEBUG, EXCEPTION, ERROR, WARNING, INFO ou NONE,
# dans l'ordre décroissant de quantité d'informations affichées.

LogLevel = EXCEPTION

# LogTraceExceptions provoque l'inclusion des traces de pile dans le journal.
# Cela rend les exceptions très verbeuses mais permet de pointer exactement
```

Exemple 15-4. Un fichier WebMacro.properties standard (suite)

```
# sur la ligne ou la méthode ayant provoqué l'erreur.

LogTraceExceptions = TRUE

# Enlever le commentaire de la ligne suivante pour journaliser dans un fichier
# à la place de stderr. Si l'erreur standard va déjà dans un journal, vous
# n'avez pas besoin de cette ligne, mais la plupart des systèmes d'exécution
# de servlets mangent simplement l'erreur standard.

# LogFile = /usr/local/webmacro/wm.log

# Définissez le modèle (relatif à TemplateDirectory) utilisé pour les erreurs.
# Vous pouvez modifier ce modèle pour personnaliser la façon dont les erreurs
# apparaissent.

ErrorTemplate = error.wm

# Le fichier continue par les fonctionnalités de "CONFIGURATION AVANCÉE"...
```

Un utilisateur avancé peut créer de multiples versions de ce fichier et passer d'une version à l'autre pour modifier rapidement et complètement le look d'un site (en changeant `TemplatePath`) ou pour simplement passer du mode développement au mode production (en changeant des paramètres comme `TemplateExpireTime`). Ce basculement peut se faire servlet par servlet car le constructeur `WM()` prend en argument un `String` optionnel spécifiant l'emplacement du fichier de configuration qu'il doit lire. Si une servlet a besoin d'accéder directement aux propriétés de ce fichier, elles sont disponibles *via* la méthode `getConfig(String key)` présente dans l'interface `WebMacro` et la classe `WMServlet`. Ainsi, le fichier de configuration contient l'équivalent des paramètres d'initialisation, excepté que ces paramètres sont liés à une configuration particulière, qui fait peut-être référence à une base de données de test pendant le développement et à une base de données de production pendant le déploiement.

Quand tout a été correctement installé, l'invocation des servlets WebMacro se fait comme l'invocation de toute autre servlet. Invoquez la servlet par l'URL *http://localhost:8080/servlet/WMHello* ou, si les servlets et les modèles sont placés sous le contexte */webmacro* comme indiqué dans le fichier *WebMacro.properties*, l'URL est *http://localhost:8080/webmacro/servlet/WMHello*. Voir la *figure 15-1*.

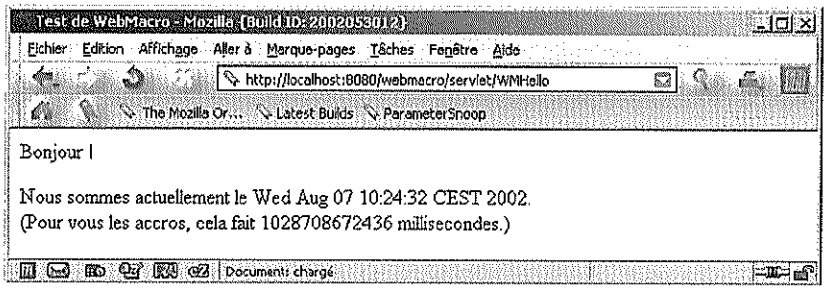


Figure 15-1. Test de WebMacro

Le langage de modèle de WebMacro

Le langage de modèle de WebMacro exploite la syntaxe de Perl et du préprocesseur C pour créer un balisage familier à beaucoup. Les variables commencent par un dollar ($) et se terminent implicitement par un espace ou un caractère de ponctuation comme <, / et ". Cela vous permet d'écrire des substitutions assez lisibles, par exemple :

```
$Request.ContextPath/servlet/WMHello
<A HREF="$url">$url</A>
```

Quand c'est nécessaire, il est possible d'utiliser des parenthèses pour entourer explicitement un nom de variable et lever une ambiguïté sur l'analyse syntaxique. Les parenthèses sont lues mais elles ne sont pas affichées.

Par exemple :

```
#set $prefix = "par"
#set $suffix = "don"
L'erreur est humaine, le $(prefix)$suffix divin.
```

Le moteur WebMacro ne se sert pas de la classe `java.beans.Introspector` pour accéder aux propriétés de l'objet ; à la place, il effectue une introspection plus complète. Quand vous écrivez `$Request.ContextPath`, le moteur WebMacro retrouve tout d'abord l'objet référencé par `$Request`, puis il tente de localiser la propriété en employant le motif de recherche suivant :

```
request.ContextPath
request.getContextPath()
request.get("ContextPath");
```

Si vous écrivez une substitution plus longue, comme `$Request.Header.Accept`, WebMacro ajoute le motif de recherche suivant à la fin de la liste :

```
request.getHeader("Accept")
```

L'affectation d'une variable suit un schéma similaire.

Avec WebMacro, vous avez la possibilité d'invoquer directement des méthodes sur tous les objets de la portée (ce que vous ne pouvez pas faire avec Tea). Par exemple, la substitution suivante trouve la liste des produits d'un inventaire, recherche le produit `hinge` et retourne son identifiant :

```
$Inventory.Products.findProduct("hinge").PartNumber
```

Cela peut se révéler utile quand les méthodes d'accès ne suivent pas le motif de nommage attendu par WebMacro. D'un autre côté, l'invocation directe des méthodes signifie qu'il ne peut y avoir d'isolation complète d'un modèle WebMacro.

Outils du contexte

Un des aspects puissants de WebMacro réside dans le fait que `WebContext` met à disposition de tous les modèles un petit nombre de variables, appelées des *outils du contexte*. Ces outils sont toujours inclus sans que la servlet appelante ait à les placer dans le contexte. La liste des outils à inclure est maintenue dans le fichier *WebMacro.properties* dans la section *WebContextTools*. Par défaut, WebMacro fournit les outils suivants :

`$Request`

Une référence à la servlet `HttpServletRequest`. Peut être utilisé pour accéder à l'en-tête `Accept` de la requête par l'instruction `$Request.Header.Accept` ou `$Request.getHeader("Accept")`.

`$Response`

Une référence à la servlet `HttpServletResponse`. Peut être utilisé pour définir l'en-tête `Content-Type` de la réponse par l'instruction `#set $Response.ContentType = "text/html"`.

`$Session`

Une référence au `HttpSession` de l'utilisateur. Peut être utilisé pour retrouver une valeur de la session par l'instruction `$Session.Attribute.Count` ou `$Session.getAttribute("Count")`.

`$Form`

Un outil pour accéder aux données de formulaire de la requête. Peut être utilisé pour retrouver le paramètre `partnumber` par l'instruction `$Form.partnumber`. C'est un raccourci de `$Request.getParameter("partnumber")` ou de `$Request.Parameter.partnumber`.

`$FormList`

Un outil pour accéder aux données de formulaire de la requête, quand les valeurs de ces données sont multiples. Il est utilisé en association à la directive `#foreach`.

`$Cookie`

Un outil pour retrouver et définir des cookies. Peut être utilisé pour définir un cookie avec l'instruction `$Cookie.set("name", "value")` ou `#set $Cookie.name = "value"`. Les cookies sont récupérés par `$Cookie.get("name").Value` ou `$Cookie.name.Value`.

`$CGI`

Un outil pour accéder aux informations de requêtes à l'aide de noms dans le style CGI. Peut être utilisé par les concepteurs de modèles maîtrisant CGI pour l'accès aux variables, comme la racine des documents : `$CGI.DOCUMENT_ROOT`.

Il est possible de fournir d'autres outils du contexte en écrivant simplement une classe qui implémente `org.webmacro.ContextTool` et en ajoutant la classe à la liste `WebContextTools`. On pourrait écrire ou télécharger des outils pour des calculs mathématiques, pour l'internationalisation, l'accès aux bases de données ou même créer des objets HTML utilisant ECS (voir le chapitre suivant). En quelque sorte, les outils du contexte sont à WebMacro ce que les bibliothèques de balises sont aux JSP.

L'*exemple 15-5* montre un outil du contexte simple qui effectue une arithmétique entière. `MathTool` se révèle souvent utile car le langage de modèle de WebMacro a été tellement simplifié qu'il n'inclut même pas les opérations arithmétiques de base.[3]

3. Cependant, du fait de la demande populaire, les opérateurs arithmétiques seront ajoutés et ils seront peut-être disponibles lorsque vous lirez ceci.

Exemple 15-5. Un outil du contexte pour l'arithmétique entière

```
import org.webmacro.*;

public class MathTool implements ContextTool {
  /**
    * Un nouvel objet sera instancié pour chaque requête en appelant cette
    * méthode. Un ContextTool est réellement une fabrique servant à créer
    * des objets utilisés dans les modèles. Certains outils peuvent
    * simplement retourner une référence sur eux-mêmes ; d'autres peuvent
    * instancier de nouveaux objets pour contenir un état propre à chaque
    * requête.
    */
  public Object init(Context c) {
    return this;
  }

  public static int add(int x, int y) {
    return x + y;
  }

  public static int subtract(int x, int y) {
    return x - y;
  }

  public static int multiply(int x, int y) {
    return x * y;
  }

  public static int divide(int x, int y) {
    return x / y;
  }

  public static int mod(int x, int y) {
    return x % y;
  }

  public static boolean lessThan(int x, int y) {
    return (x < y);
  }

  public static boolean greaterThan(int x, int y) {
    return (x > y);
  }
}
```

Une fois cet outil ajouté à la liste `WebContextTools` (n'oubliez pas cette étape), tous les modèles peuvent effectuer des mathématiques basiques. Par exemple, l'affichage de l'année d'un objet `Date`, il faut ajouter 1900 à la propriété `Year` :

```
L'année courante est $Math.add($date.Year, 1900).
```

Directives de WebMacro

Les directives sont des instructions WebMacro qui effectuent une certaine opération, comme inclure du texte selon une condition ou répéter un bloc au sein d'un modèle. La liste des directives disponibles se trouve dans le fichier *WebMacro.properties* dans la propriété `Directives`. Les directives commencent toutes par un caractère #. Certaines opèrent sur un bloc, ces blocs pouvant être délimités par des accolades, { et }, ou par les mots-clés `#begin` et `#end`. Les mots-clés ne sont pas aussi pratiques à saisir que les accolades mais ils génèrent moins de perturbations lorsque du JavaScript est inclus.

Il existe par défaut sept directives couramment utilisées. Comme pour les outils du contexte, vous pouvez écrire vos propres directives, même si c'est plutôt exceptionnel. Examinons les sept directives par défaut.

#if

```
#if (condition) { ... } #else { ... }
```

La directive `#if`, seule ou associée à la directive `#else`, sert à inclure du texte de façon conditionnelle. Le texte est inclus si la condition est vraie, sinon il ne l'est pas. Une condition est vraie si sa valeur non nulle n'est pas le booléen `false`. La condition de `#if` peut employer les opérateurs booléens classiques &&, || et !, ainsi que les parenthèses pour indiquer un ordre. Des objets sont comparés dans la condition à l'aide des opérateurs == et !=, qui invoquent en fait la méthode `.equals()` pour effectuer la comparaison. Il n'existe pas d'opérateur inférieur-à ou plus-grand-que car les objets n'ont pas l'obligation d'offrir ce type de comparaison. `MathTool` inclut ces comparaisons pour les entiers.

```
#if ($Client.doitArgent() && $Client.Nom != "Jason") {
  Payez, sinon...
}
#else {
  Bonjour !
}
```

#set

```
#set $propriété = valeur
```

La directive `#set` attribue une valeur à la variable indiquée ou à une propriété. La propriété doit pouvoir être retrouvée à l'aide de la logique d'introspection de WebMacro et elle doit être publiquement modifiable d'une manière ou d'une autre. Si la variable n'est pas dans la portée, une nouvelle variable est créée et prend la valeur indiquée. Les variables sont implicitement du type `String` ou `Integer`. Par défaut, le type `String` est employé sauf si la valeur peut être convertie en un `Integer` et si elle n'est pas entourée de guillemets doubles. Pour plus de facilité, les valeurs `Integer` peuvent être converties en valeurs `String` si nécessaire. Des tableaux peuvent aussi être définis à l'aide de crochets. Utilisez une barre oblique inverse pour appliquer l'échappement sur un dollar :

```
#set $num1 = 4
#set $num2 = 7
#set $prix = "90"
```

```
#set $facture = "Vous devez \$$price"
#set $note = "$num1 achats, il y a $num2 an(s)..."
#set $tout = [ $num1, $num2, $quote ]  ## syntaxe pour les tableaux.
```

#foreach

```
#foreach $chose in $liste { ... }
```

La directive #foreach itère sur une liste, incluant son bloc de sortie une fois pour chaque élément de la liste. À chaque tour de boucle, la variable $chose prend la valeur du terme suivant de $liste. La liste peut être un tableau déclaré dans WebMacro ou un objet Java satisfaisant une des conditions suivantes (recherchées dans cet ordre par l'introspection) :

1. L'objet est lui-même un tableau.
2. L'objet est lui-même un `Iterator`.
3. L'objet est lui-même un `Enumeration`.
4. L'objet a une méthode `Iterator iterator()`.
5. L'objet a une méthode `Enumeration elements()`.

Le code suivant affiche les locales préférées du client (si elles existent) :

```
#set $tableau = $Request.Locales
<UL>
#foreach $element in $tableau {
  <LI>$item
}
</UL>
```

#parse

```
#parse fichier
```

La directive #parse inclut le contenu du fichier cible à l'emplacement de la directive et l'analyse comme s'il faisait partie du modèle courant. On dispose ainsi d'un mécanisme pratique pour inclure du code de modèle commun à plusieurs fichiers modèle. Le fichier peut être indiqué avec un chemin absolu ou un chemin relatif à celui donné par `TemplatePath` :

```
#set $titre = "Page de titre"
#parse "header.wm"  ## La variable $titre est visible dans header.wm.
```

#include

```
#include url
```

Le directive #include inclut le contenu de l'URL cible à l'emplacement de la directive mais n'essaie pas d'en analyser le contenu. On dispose ainsi d'un mécanisme pratique pour inclure du JavaScript ou d'autres choses dont la syntaxe peut entrer en conflit avec WebMacro. Si l'URL ne définit pas le protocole, `file:` est pris par défaut :

```
#include "http://webmacro.org/CREDITS"  ## Inclusion de texte brut.
```

#param

```
#param $nom = valeur
```

La directive #param désigne les valeurs dans le modèle qui peuvent être examinées par la servlet appelant ce modèle. Ainsi, l'auteur du modèle dispose d'un moyen pour passer des informations au programmeur Java qui peut s'en servir pour déterminer le type des informations à placer dans le contexte :

```
#param $auteur = "Susan Kelley"
#param $requis = [ "utilisateur", "document", "session" ]
```

La servlet retrouve la valeur en appelant `Object getParam(String)` sur l'objet `Template`. La méthode retourne un `String` ou un `Integer` si une valeur existe et un `Object[]` contenant des objets `String` et/ou `Integer` si la valeur est une liste :

```
String auteur = (String) tmpl.getParam("auteur");
Object[] requis = (Object[]) tmpl.getParam("requis");
```

#use

```
#use 'parser'
#begin
  ...
#end
```

Enfin, la directive #use permet l'analyse d'un bloc de texte modèle (délimité par les marqueurs #begin et #end) par un autre analyseur. Cette fonctionnalité autorise des améliorations ou même le remplacement complet de la syntaxe de WebMacro. En général, elle n'est pas employée pour de si grands desseins. Ses deux utilisations les plus courantes sont pour inclure un bloc de texte (avec l'analyseur `text`) ou pour supprimer un bloc de texte (avec l'analyseur `null`) :

```
#use 'text'
#begin
  Le texte ici présent est inclus tel quel.
  ## Les commentaires aussi car leur format est une caractéristique
  ## de l'analyseur.
#end

#use 'null'
#begin
  Le texte ici présent peut êre considéré comme un "commentaire".
#end
```

Le fichier *WebMacro.properties* maintient dans la propriété `Parsers` la liste de tous les analyseurs disponibles par défaut.[4]

4. Le mécanisme d'analyseur syntaxique connectable peut être remplacé par des directives ciblées. Par exemple, `#use 'text'` peut être remplacé par `#text` et `#use 'null'` par `#comment`.

Modèles WebMacro

Pour montrer l'emploi des outils du contexte et des directives disponibles aux modèles WebMacro, l'*exemple 15-6* montre un modèle simple qui affiche des informations sur la requête.

Exemple 15-6. Analyse de la requête avec WebMacro

```
## snoop.wm

#set $Response.ContentType = "text/html"

<HTML><HEAD><TITLE>Analysons !</TITLE></HEAD>
<BODY>

## Un modèle d'analyse pour se familiariser avec WebMacro.

<H1>Informations diverses</H1>
Cha&icirc;ne d'interrogation : $Request.QueryString<BR>
Utilisateur distant : $Request.RemoteUser<BR>

## WebMacro ne reconnaît pas encore les propriétés isXXX() ; nous devons
## employer un appel de méthode. Il ne dispose pas non plus de elseif (il
## sera ajouté).

#if ($Request.isRequestedSessionIdFromCookie())  {
  Vous &ecirc;tes dans une session bas&eacute;e sur les cookies !
}
#else {
  #if ($Request.isRequestedSessionIdFromURL()) {
    Vous &ecirc;tes dans une session bas&eacute;e sur la
    r&eacute;&eacute;criture d'URL !
  }
  #else {
    Vous n'&ecirc;tes pas dans une session.
  }
}

<H1>Informations sur les param&egrave;tres</H1>
#foreach $paramName in $Request.ParameterNames {
  $paramName : $Request.getParameter($paramName) <BR>
}

<H1>Informations sur les en-t&ecirc;tes</H1>
#foreach $headerName in $Request.HeaderNames {
  $headerName : $Request.getHeader($headerName) <BR>
}

<H1>Informations sur les cookies</H1>
#foreach $cookie in $Request.Cookies {
```

Exemple 15-6. Analyse de la requête avec WebMacro (suite)

```
  $cookie.Name: $cookie.Value <BR>
}

</BODY></HTML>
```

La *figure 15-2* montre une copie d'écran.

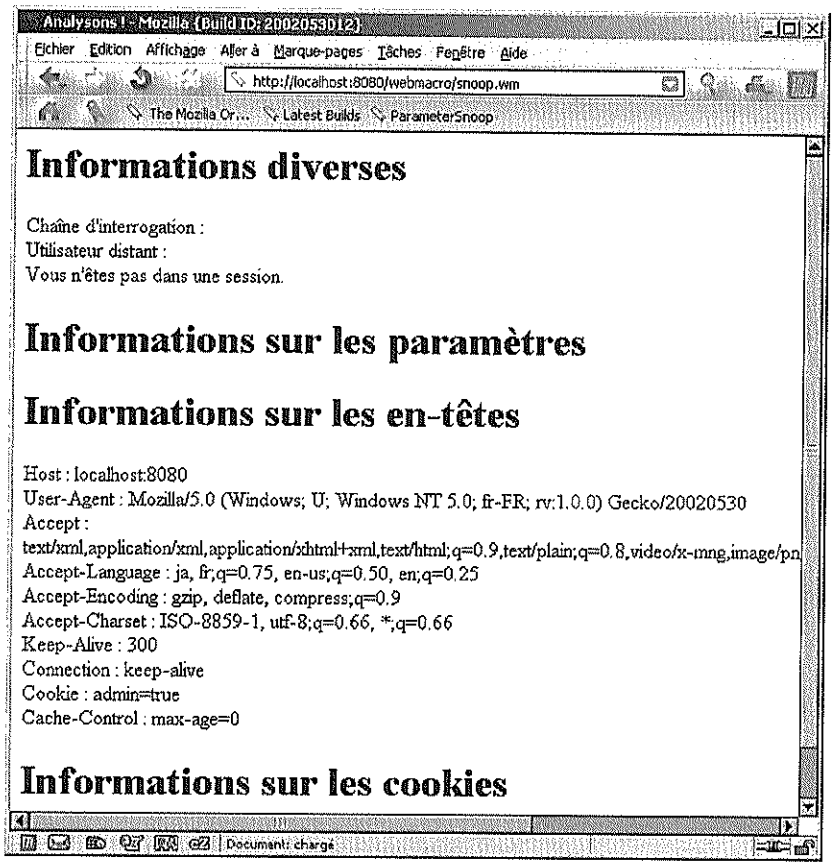


Figure 15-2. Analyse avec WebMacro

Ce modèle montre l'emploi de la directive `#set` pour définir le type de contenu de la réponse. Il utilise les directives `#if` et `#else` pour déterminer si le client a rejoint une session en utilisant un cookie ou la réécriture d'URL, ou s'il ne fait pas partie d'une session. Il utilise la directive `#foreach` pour parcourir les paramètres, les en-têtes et les cookies présents dans la requête.

Notez les quelques astuces suivantes. Premièrement, une méthode d'accès du bean suivant le motif de nommage `is`*`Propriété`*`()` n'est pas trouvée automatiquement par WebMacro et elle doit être appelée explicitement par une invocation de méthode. Deuxièmement, puisque la directive `#elseif` n'existe pas, une construction if/elseif/else doit être réalisée par des imbrications.

Une servlet MacroView réutilisable

Le modèle de l'*exemple 15-6* peut être considéré « autonome » car il n'utilise aucune variable fournie par une servlet. Les modèles autonomes ne peuvent être invoqués directement et une servlet doit les appeler. Mais puisque le modèle n'a pas de contraintes spécifiques, cette servlet peut être une servlet MacroView générique et réutilisable, comme montré à l'*exemple 15-7*.

Exemple 15-7. Une servlet WebMacro générique réutilisable

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.webmacro.*;
import org.webmacro.servlet.*;
import org.webmacro.engine.*;
import org.webmacro.broker.*;

// Étendre com.oreilly.servlet.CacheHttpServlet peut améliorer les temps
// de réponse.

public class MacroView extends HttpServlet {

  WebMacro wm;  // Lien principal avec WebMacro.

  public void init() throws ServletException {
    try {
      wm = new WM();
    } catch (InitException e) {
      throw new ServletException(e);
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                  throws ServletException, IOException {
    FastWriter out = new FastWriter(res.getOutputStream(),
                                    res.getCharacterEncoding());

    // Le nom du modèle vient dans l'information de chemin supplémentaire
    //   /servlet/MacroView/templ.wm
    // ou comme chemin de servlet via une règle *.wm
    //   /templ.wm
    String template = req.getPathInfo();
    if (template == null) {
      template = req.getServletPath();
      template = template.substring(1);  // Enlever le "/" final.
    }

    // Si template vaut toujours null, nous avons un problème.
```

Exemple 15-7. Une servlet WebMacro générique réutilisable (suite)

```
    if (template == null) {
      throw new ServletException("Aucun modèle indiqué dans l'information "+
        "de chemin supplémentaire ou dans le chemin de servlet");
    }

    try {
      Template tmpl = wm.getTemplate(template);
      WebContext context = wm.getWebContext(req, res);
      tmpl.write(out, context);
    } catch (WebMacroException e) {
      throw new ServletException(e);
    }
    finally {
      out.flush();
    }
  }

  public void destroy() {
    super.destroy();
    if (wm != null) wm.destroy();
  }
}
```

Cette servlet appelle tout nom de modèle passé dans l'information de chemin supplémentaire, ou, si elle n'est pas présente, tout nom de modèle désigné par le chemin de servlet (par le biais d'une correspondance d'extension de fichier). Il est pratique d'enregistrer cette servlet pour qu'elle gère toutes les requêtes **.wm* en utilisant l'extrait du fichier *web.xml* montré à l'*exemple 15-8*. Cela permet aux fichiers modèle de WebMacro d'apparaître comme demandés directement *via* des URL telles que */snoop.wm* ou */webmacro/snoop.wm*, alors qu'en réalité la servlet `MacroView` réalise le traitement en coulisses.

*Exemple 15-8. Enregistrement de MacroView pour gérer les fichiers *.wm*

```
<servlet>
    <servlet-name>
        mv
    </servlet-name>
    <servlet-class>
        MacroView
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>
        mv
    </servlet-name>
    <url-pattern>
        *.wm
    </url-pattern>
</servlet-mapping>
```

Il pourrait être utile d'étendre la servlet `MacroView` afin qu'elle applique une certaine logique métier sur chaque requête et peut-être qu'elle rende disponible un ensemble d'objets communs dans le contexte pour tous les modèles système. Une telle servlet, associée aux outils du contexte précédents, permet aux modèles WebMacro d'opérer dans un modèle similaire à Tea (en fait, c'est ainsi qu'AltaVista utilise WebMacro).

Traitement de modèles

WebMacro effectue une grande partie de l'analyse des modèles en coulisses, si bien que les programmeurs et les concepteurs n'ont en général pas besoin d'en connaître les mécanismes pour en obtenir le meilleur. WebMacro analyse un modèle à sa première utilisation et en stocke ensuite une représentation en mémoire pour une exécution rapide. WebMacro recharge et analyse à nouveau automatiquement le contenu du modèle si son fichier change mais, pour des raisons d'efficacité, il contrôle les estampilles selon un intervalle de temps indiqué dans la propriété `TemplateExpireTime` de *WebMacro.properties*. Par défaut, l'intervalle est de 0 millisecondes afin que les estampilles soient testées à chaque requête. C'est pratique pour le développement, mais pensez à augmenter cet intervalle pour un système en production. Pour les utilisateurs avancés, les objets de modèle ont une méthode `parse()` qui force le chargement et l'analyse. Vous pouvez l'utiliser pour analyser tous vos modèles au lancement ou pour forcer une nouvelle analyse si vous savez qu'un modèle a été modifié.

Quand des modèles contiennent des erreurs, diverses choses peuvent se produire. Si une variable ou une propriété référencée a une valeur `null` ou si elle n'existe pas, WebMacro traite ce cas de substitution manquée comme une erreur non catastrophique. Il génère la page de son mieux et écrit un message `WARN` dans le journal. Il place aussi dans la page générée un commentaire HTML/XML à l'emplacement de l'erreur (cette approche pose problème si la page générée n'est pas en HTML ou en XML) :

```
<!-- warning: attempt to write out undefined variable Request.ContentType:
java.lang.NullPointerException -->.
```

Si un modèle contient une erreur de syntaxe, WebMacro considère qu'il s'agit d'une erreur catastrophique. Il écrit un message `ERROR` dans le journal et affiche dans la page générée une description de l'erreur.[5] Pour certains types d'erreurs, comme un modèle non trouvé par une servlet, WebMacro dispose d'un modèle d'erreur par défaut qu'il peut afficher. Ce modèle se configure à l'aide de la propriété `ErrorTemplate` dans *WebMacro.properties*.

Bien entendu, la majeure partie du traitement des erreurs dans un site WebMacro — gérer les paramètres invalides, les bases de données erronées, les fichiers manquants et toutes les autres erreurs qui doivent être testées avant de passer le contrôle au modèle — doit se passer dans le code de la servlet. WebMacro fonctionne bien sur ce point. Une servlet peut utiliser un ensemble de modèles standard de WebMacro pour afficher ces cas d'erreur. Elle choisit le modèle auquel passer le contrôle et ce qu'il faut inclure dans

5. Il y a un bogue dans WebMacro 0.94 (la version avec laquelle le code de ce chapitre a été testé) qui conduit à la génération d'une page vide au lieu d'une page contenant la description de l'erreur lorsqu'un modèle contient une erreur de syntaxe. Il sera corrigé dans les prochaines versions.

le contexte en fonction de l'erreur. Par exemple, si une erreur de base de données survient, le contrôle peut être passé à un fichier *sqlException.wm* accompagné de la trace de l'exception à afficher.

Une application outil

Pour conclure notre présentation de WebMacro, regardons comment nous pourrions écrire l'application outil du chapitre précédent à l'aide de WebMacro en place de Tea. Elle est plus facile à comprendre si nous commençons par examiner le modèle, montré à l'*exemple 15-9*.

Exemple 15-9. Le modèle WebMacro de l'application outil

```
## toolview.wm

#set $Response.ContentType = "text/html"

#set $title = "Liste des outils"
#set $deck = "Une liste des outils de création de contenu"
#set $desc = "Sans les outils, les gens ne sont rien de plus que des  \
              animaux. Et plutôt faibles, en plus. Voici une liste \
              d'outils de création de contenu basé sur les servlets que \
              vous pouvez utiliser afin de ne plus être une mauviette."

#parse "header.wm"

## Définir une valeur lue par la servlet.
#param $defaultState = "LIVE"

#foreach $tool in $tools {

  <HR SIZE=2 ALIGN=LEFT>

  <H3>
  $tool.Name

  #if ($tool.isNewWithin(45)) {
    <FONT COLOR=#FF0000><B> (New!) </B></FONT>
  }
  #else {
    #if ($tool.isUpdatedWithin(45)) {
      <FONT COLOR=#ff0000><B> (Updated!) </B></FONT>
    }
  }
  </H3>
  <A HREF="$tool.HomeURL">$tool.HomeURL</A><BR>

  $tool.Comments

}

   #parse "footer.wm"
```

Le début de ce modèle définit des valeurs pour le titre, le sujet et la description de la page (le caractère \ sert à continuer une ligne dans la description). Ces variables sont utilisées dans le modèle *header.wm* pour créer la structure de la page entourant la liste des outils. Les variables sont aussi visibles dans le modèle *footer.wm* analysé à la fin du modèle, même s'il ne s'en sert pas. Les modèles d'en-tête et de base de page sont montrés à l'*exemple 15-10* et à l'*exemple 15-11*, respectivement.

Le modèle *toolview.wm* utilise la directive `#param` pour donner au paramètre de modèle `defaultState` la valeur constante `LIVE`. La servlet peut ainsi appeler ce modèle pour déterminer quels outils afficher par défaut dans cette page.

Ensuite, le modèle utilise une directive `#foreach` pour itérer sur la liste `tools` passée par la servlet. La variable `tools` peut être un tableau, un `Iterator`, un `Enumeration` ou un objet de type `Vector` ou `List` avec une méthode `iterator()` ou une méthode `elements()` — le modèle s'en fiche. Pour chaque outil, le modèle affiche son nom, s'il est nouveau ou mis à jour, son URL et ses commentaires. Le modèle invoque les méthodes `isNewWithin(int)` et `isUpdatedWithin(int)` pour déterminer si le contenu doit être considéré comme nouveau ou mis à jour. Le modèle pourrait accéder à `$tool.CreatedAgeInDays` et à `$tool.ModifiedAgeInDays`, mais il faudrait alors le support de l'outil du contexte `MathTool` de l'*exemple 15-5* pour effectuer les comparaisons.

Exemple 15-10. Le fichier header.wm

```
## header.wm

<HTML><HEAD><TITLE>$title</TITLE></HEAD>

<BODY BGCOLOR="#FFFFFF" BACKGROUND="/images/background.gif"
      LINK="#003333" ALINK="#669999" VLINK="#333333">

<IMG SRC="/images/banner.gif" WIDTH=600 HEIGHT=87 BORDER=0><BR>

<TABLE>
<TR>
<TD WIDTH=125 ROWS=10 VALIGN=TOP>
 <BR><BR><BR>
 <FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#FF0000">
 <A HREF="/index.html">Accueil</A><BR>
 <A HREF="/hosting.html">Hébergement</A><BR>
 <A HREF="/engines.html">Moteurs</A><BR>
 </FONT>
</TD>

<TD WIDTH=475>

  <TABLE CELLPADDING=5><TR><TD WIDTH=600 BGCOLOR="#006699" VALIGN=TOP>
  <B><FONT FACE="Arial,Helvetica" SIZE="+2">
  $title
  </FONT></B>
  </TD></TR></TABLE>
```

Exemple 15-10. Le fichier header.wm (suite)

```
  <B><FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#003366">
  $deck
  </FONT></B><P>

  <P>

  <FONT FACE="Arial,Helvetica">

  $desc
```

Exemple 15-11. Le fichier footer.wm

```
## footer.wm

 </FONT>

</TD>
</TR>

<TR>
<TD></TD>
<TD WIDTH=475 ALIGN=CENTER COLSPAN=3>
<HR>
<FONT FACE="Arial,Helvetica">
<A HREF="/index.html">Accueil</A>  
<A HREF="/hosting.html">Hébergement</A>  
<A HREF="/engines.html">Moteurs</A>  <P>
</FONT>

<TABLE WIDTH=100%>
<TR>
<TD WIDTH=260 ALIGN=LEFT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
<A HREF="/copyright.html">Copyright</A> &copy; 2000 Jason Hunter<BR>
Tous droits réservés.</TD>
<TD WIDTH=5></FONT></TD>
<TD WIDTH=230 ALIGN=RIGHT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
Contact : <A HREF="mailto:webmaster@servlets.com">webmaster@servlets.com</a>
</FONT></TD>
</TR>
</TABLE>

</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Le code de la classe Tool est le même que celui de l'application Tea, même si, avec WebMacro, les méthodes d'accès du bean ne sont pas nécessaires et si isNewWithin(int) et isUpdatedWithin(int) sont vues par le modèle. Le code de la classe ToolServlet est montré à l'*exemple 15-12*.

Exemple 15-12. La servlet pilotant l'application outil

```
import org.webmacro.*;
import org.webmacro.servlet.*;
import org.webmacro.util.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;

public class ToolServlet extends WMServlet {

  private Log log;
  private Tool[] tools;

  public void start() throws ServletException {
    // Charger les données sur les outils dans l'initialisation pour plus
    // de simplicité.
    String toolsFile = getInitParameter("toolsFile"); // De web.xml.
    if (toolsFile == null) {
      throw new ServletException(
        "Un fichier de données sur les outils doit être spécifié dans " +
        "le paramètre d'initialisation toolsFile");
    }
    log = new Log(getServletName(), "Journal de débogage de l'exemple outil");
    log.debug("Chargement des outils à partir de " + toolsFile);
    try {
      tools = Tool.loadTools(toolsFile);
      if (tools.length == 0) {
        log.warning("Aucun outil trouvé dans " + toolsFile);
      }
      else {
        log.info(tools.length + " outil(s) trouvé(s) dans " + toolsFile);
      }
    }
    catch (Exception e) {
      log.error(e);
      throw new ServletException(e);
    }
  }

  // La création d'un contexte fournit des fonctions accessibles à partir
  // des modèles.
  public Template handle(WebContext context) throws HandlerException {
    // On passe souvent la requête, la réponse et l'application même si tous
    // ces objets ne sont pas utilisés, car ils pourraient l'être plus tard.
```

Exemple 15-12. La servlet pilotant l'application outil (suite)

```
    try {
      Template view = getTemplate("toolview.wm");
      String state = context.getRequest().getParameter("state");
      if (state == null) {
        state = (String)view.getParam("defaultState");
      }

      if (state == null) {
        context.put("tools", getTools());
      }
      else {
        context.put("tools", getTools(state));
      }
      return view;
    }
    catch (WebMacroException e) {
      log.exception(e);
      throw new HandlerException(e.getMessage());
    }
    catch (IOException e) {
      log.exception(e);
      throw new HandlerException(e.getMessage());
    }
  }

  public Tool[] getTools() {
    return tools;
  }

  public Tool[] getTools(String state) {
    List list = new LinkedList();
    for (int i = 0; i < tools.length; i++) {
      if (tools[i].getStateFlag().equalsIgnoreCase(state)) {
        list.add(tools[i]);
      }
    }
    return (Tool[]) list.toArray(new Tool[0]);
  }
}
```

La servlet étend `WMServlet` et définit donc une méthode `start()` au lieu de `init()`, ainsi qu'une méthode `handle()` en place de `doGet()` et de `doPost()`. La méthode `start()` retrouve le paramètre d'initialisation `toolsFile` et charge les données sur les outils se trouvant dans le fichier à l'aide de `Tool.loadTools(toolsFile)`. La méthode crée aussi une instance de `Log` et l'utilise pour indiquer sa progression en écrivant des messages d'information, de débogage, d'avertissement et d'erreur.

La méthode `handle()` charge le modèle *toolview.wm* puis lit le paramètre `state` de la requête pour déterminer les outils à afficher. Si ce paramètre n'existe pas, la servlet utilise

la valeur par défaut indiquée dans le modèle lui-même. Elle place ensuite dans le WebContext la liste des outils avec tools pour nom de variable et elle retourne le modèle. Toute exception lancée est journalisée et passée à un HandlerException.

Les méthodes getTools() en fin de servlet ne sont que des méthodes utilitaires. Elles ne sont pas visibles du modèle, même si elles le pourraient en incluant dans le contexte un objet lookup contenant ces méthodes. Ensuite, un modèle pourrait effectuer sa propre gestion du paramètre d'état :

```
#set $state = $Form.state;
#if (!state) {
  #set $tools = $lookup.getTools("live")
}
#else {
  #set $tools = $lookup.getTools($state)
}
```

Vous avez le choix entre cette approche de type *pull* et le modèle de type *push* présenté précédemment ; c'est une question de préférence. WebMacro a tendance à encourager le modèle push et Tea le modèle pull ; encore que tous deux supportent facilement les deux modèles.

Vous pouvez utiliser la classe com.oreilly.servlet.CacheHttpServlet du chapitre 3, *Cycle de vie d'une servlet*, pour mettre en cache la sortie et améliorer les performances des modèles. La technique fonctionne mieux avec les modèles dont la génération de la sortie prend un temps significatif mais dont le contenu change peu souvent, comme les modèles qui affichent des résultats issus d'une base de données. Pour exploiter la fonction de cache, vous devez écrire votre servlet de manière à ce qu'elle étende CacheHttpServlet (vous devrez vous passer de l'aide de WMServlet et gérer manuellement les modèles) et implémenter getLastModified(). Si les données proviennent d'une base de données, la méthode getLastModified() peut renvoyer la date de dernière mise à jour de la base (dans une variable de contexte affectée par la servlet effectuant la mise à jour) ou, si la date de mise à jour ne peut être déterminée, la méthode peut simplement retourner la date courante modulo une certaine valeur afin que les mises à jour se fassent à intervalle régulier. Si les données proviennent d'un fichier, la méthode peut retourner la date à laquelle le fichier a été mis à jour pour la dernière fois (peut-être en ne faisant cette vérification que de temps à autre pour des raisons d'efficacité).

Filtres

Au moment de l'écriture de ce livre, WebMacro a juste ouvert son processus de développement à un public plus large et il est passé d'une licence GPL assez restrictive à une licence de type Apache moins restrictive. Du fait de l'augmentation des ressources de développement, plusieurs améliorations intéressantes et utiles sont en cours.

L'amélioration la plus importante concerne le mécanisme de filtre connectable. L'idée est qu'une variable passe au travers d'un filtre au moment de la sortie, le filtre contrôlant l'affichage de la variable. Les filtres standard vont sans doute appliquer les traitements suivants :

Échappement HTML

Il s'agit de filtrer les champs de commentaires entrés par l'utilisateur afin d'empêcher les caractères spéciaux de provoquer des problèmes accidentellement ou intentionnellement.

Localisation

Il s'agit de passer une date au travers d'une logique d'affichage localisé.

Réécriture d'URL

Il s'agit de filtrer une URL afin d'y ajouter la valeur de suivi de session `jsessionid`.

Affichage des valeurs nulles

Il s'agit de filtrer l'affichage des variables `null` pour afficher un message d'erreur ou pour l'ignorer silencieusement.

Filtres personnalisés

Les filtres personnalisés ont un avenir certain. Par exemple, une variable `client` pourrait être passée au travers d'un modèle personnalisé conçu de manière à afficher la variable et ses propriétés d'après un fichier de modèle externe. Le modèle client pourrait même filtrer l'adresse du client à l'aide d'un modèle d'adresse externe. Cela facilite la réutilisation et la personnalisation de la logique d'affichage. Les filtres peuvent aussi accepter leur chaînage. Par exemple, cela permettrait d'appliquer l'échappement et la localisation à une variable, tout en définissant une valeur de sortie par défaut si la variable était `null`.

Les utilisations possibles des filtres sont encore purement dans l'imagination des gens. Il sera intéressant de voir lesquelles seront effectivement répandues.

16

Element Construction Set

L'approche à la génération de contenu du paquetage ECS (*Element Construction Set*) est radicalement différente de celle des JSP, de Tea et de WebMacro. ECS fait abstraction du texte HTML et voit HTML comme un autre ensemble d'objets Java. Dans ECS, une page web est manipulée comme un objet qui contient d'autres objets HTML (comme des listes et des tableaux) qui peuvent contenir d'autres objets HTML (comme des éléments de liste et des cellules de tableaux). Ce modèle de « génération HTML orientée objet » s'avère très puissante, mais aussi très orientée programmeur.

Stephan Nagy et Jon Stevens ont développé ECS et l'ont rendu en *open source* au sein du Java Apache Project, bien sûr sous licence Apache. Ils ont modélisé la bibliothèque d'après le produit commercial htmlKona de WebLogic, un produit dont le support a disparu lorsque WebLogic a été racheté par BEA Systems. Nous présentons dans ce chapitre d'ECS Version 1.3.3, disponible à *http://jakarta.apache.org/ecs*.[1]

Composants d'une page sous forme d'objets

ECS inclut des classes pour toutes les constructions HTML 4.0. L'*exemple 16-1* montre comment construire avec ECS une page HTML simple.

Exemple 16-1. Une page vue comme un ensemble d'objets

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.ecs.*;
import org.apache.ecs.html.*;
```

1. *N.d.T.* : au moment de la traduction de ce livre, la version courante d'ECS est la version 1.4.1.

Exemple 16-1. Une page vue comme un ensemble d'objets (suite)

```
public class ECSHello extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    Document doc = new Document();
    doc.appendTitle("Test d'ECS");
    doc.appendBody(new Big("Bonjour !"))
       .appendBody(new P())
       .appendBody("Nous sommes actuellement le " + new Date());
    doc.output(out);
  }
}
```

Notez comment toutes les balises HTML ont été remplacées par des objets. Cette servlet crée un nouvel objet `Document` qui représente la page web à renvoyer. Ensuite, elle ajoute un titre « Test d'ECS » à la page, un corps de page contenant un gros « Bonjour ! », une coupure de paragraphe et la date courante. Enfin, la servlet sort la page sur son `PrintWriter`. C'est ainsi que fonctionne la génération HTML orientée objet : obtenir un objet `Document`, y ajouter les objets représentant les composants et l'envoyer au client.

Pour exécuter cette servlet, vous devez installer ECS en plaçant le fichier JAR d'ECS dans le chemin de classes de votre serveur (ou dans le répertoire *WEB-INF/lib* de votre application web). Pour ECS 1.3.3, le fichier JAR s'appelle *ecs-1.3.3.jar*. Vous devrez sans doute redémarrrez votre serveur pour qu'il trouve ce nouveau JAR. Une fois ECS installé, vous pouvez invoquer la servlet de la manière habituelle et elle doit générer une sortie semblable à la suivante :

```
<html><head><title>Test d'ECS</title></head><body>
<big>Bonjour !</big><p>Nous sommes actuellement le
Thu Aug 08 10:11:14 CEST 2002</body></html>
```

Par défaut, la sortie d'ECS apparaît sur une seule ligne, sans indentation ni retour chariot. Cela économise de la bande passante lors des communications avec le client. Si vous voulez une version plus lisible, modifiez le fichier *ecs.properties* qui vient avec la distribution pour donner la valeur `true` à `pretty_print`. Puis, la partie astucieuse : assurez-vous que votre fichier *ecs.properties* se trouve dans le chemin de classes du serveur *avant* le fichier JAR d'ECS afin que votre fichier modifié remplace celui inclus dans le JAR. Le fichier *ecs.properties* étant recherché dans *org/apache/ecs*, il ne doit donc pas être placé directement dans le chemin de classes mais dans un sous-répertoire *org/apache/ecs* d'un répertoire dans le chemin de classes (par exemple, *WEB-INF/classes/org/apache/ecs/ecs.properties*).

La servlet importe deux paquetages ECS : `org.apache.ecs` pour les classes ECS fondamentales et `org.apache.ecs.html` pour les classes spécifiques à HTML. (Il existe d'autres paquetages pour XML, WML et RTF.)[2] Le paquetage `org.apache.ecs.html`

2. Si vous êtes intéressé par la génération de XML à partir de Java, vous devriez examiner JDOM (*http://jdom.org*) car son intégration aux technologies XML est bien meilleure.

contient à peu près une centaine de classes représentant tous les composants HTML 4.0. La majorité des classes HTML ont des noms correspondant aux noms des balises HTML qu'elles représentent : `Big`, `Small`, `P`, `Table`, `TR`, `TD`, `TH`, `H1`, `H2`, `H3`, `Frame`, `A`, `Head`, `Body`, etc. Chaque classe HTML a des méthodes permettant de configurer le composant. Par exemple, la classe `TD` offre une méthode `setBackground(String url)` qui définit la couleur de fond de la cellule de la table. La classe `Body` offre aussi une méthode similaire pour définir la couleur de fond de la page entière. Aucun autre composant ECS n'offre une méthode `setBackground()` car aucun autre composant ne peut définir sa couleur de fond. Cela permet à ECS de garantir que les documents créés par programmation sont toujours du HTML bien formé.

Pour ajouter des éléments au corps, la servlet utilise le chaînage de méthodes, dans lequel plusieurs méthodes sont invoquées sur le même objet. C'est très courant avec ECS, par exemple, pour construire un tableau :

```
Table tab = new Table()
            .setCellPadding(0)
            .setCellSpacing(0);
```

Les espaces ne sont pas importants. Le code précédent est équivalent à :

```
Table tab = new Table().setCellPadding(0).setCellSpacing(0);
```

Ce chaînage est possible car les méthodes `set` et `append` retournent une référence à l'objet sur lequel elles ont été invoquées ; cette référence est utilisée pour invoquer la méthode suivante. Cette astuce est très pratique avec ECS.

Afficher un ensemble de résultats

L'emploi d'ECS pour la création de pages complètes a diminué lorsque les technologies de modèles basées sur les servlets se sont améliorées. Il demande tout simplement trop de travail pour créer dynamiquement ce qui n'est principalement que du contenu statique. Cependant, ECS a encore sa place. Il fonctionne très bien pour les parties *extrêmement* dynamiques d'une page, où vous avez besoin de toute la puissance de Java pour déterminer le contenu à produire. Comme le dit Jon Stevens, un de ses créateur, « Utilisez ECS là où vous utiliseriez sinon des `out.println()`. »

Par exemple, examinons une application web qui laisse les clients exécuter des requêtes *ad hoc* sur une base de données. Peut-être souhaitez-vous implémenter un système de suivi de bogues et peut-être voulez-vous qu'une servlet donne aux utilisateurs avancés la possibilité d'exécuter leurs propres requêtes sur les données (avec une connexion en lecture seule). ECS est très bien adapté à l'écriture de pages qui affiche les résultats issus d'une base de données et affichés dans un tableau généré par programmation. L'*exemple 16-2* montre un composant de visualisation simple, `ResultSet`. Il vient de la classe `HtmlSQLResult` du chapitre 9, *Se connecter aux bases de données*, qui utilisait `out.println()`. En utilisant ECS à la place de `out.println()`, le code devient plus simple et, comme nous le verrons à la prochaine section, plus personnalisable.

Exemple 16-2. Un tableau des résultats amélioré

```
import java.io.*;
import java.sql.*;
import java.util.*;
```

Exemple 16-2. Un tableau des résultats amélioré (suite)

```
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class SimpleResultSetTable extends Table {

  public SimpleResultSetTable(ResultSet rs) throws SQLException {
    setBorder(1);

    ResultSetMetaData rsmd = rs.getMetaData();
    int colCount = rsmd.getColumnCount();

    TR row = new TR();
    for (int i = 1; i <= colCount; i++) {
      addElement(new TH().addElement(rsmd.getColumnName(i)));
    }
    addElement(row);

    while (rs.next()) {
      row = new TR();
      for (int i = 1; i <= colCount; i++) {
        addElement(new TD().addElement(rs.getString(i)));
      }
      addElement(row);
    }
  }
}
```

Cette classe étend `org.apache.ecs.html.Table` et représente donc un élément `<TABLE>` de HTML. Elle réalise tout son travail dans son constructeur, lisant le `ResultSet` et le `ResultSetMetaData` associé pour générer un simple tableau de résultats. Le constructeur appelle d'abord `setBorder(1)` pour définir l'attribut de bord du tableau. Ensuite, il crée une ligne (`TR`) remplie avec des éléments d'en-tête (`TH`) chacun affichant le nom de colonne pris dans les métadonnées. Enfin, le constructeur boucle sur l'ensemble de résultats et crée une ligne pour chaque entrée, la remplissant avec des éléments de données de tableau (`TD`) contenant une représentation simple, sous forme de `String`, des données de l'ensemble de résultats.

La classe `SimpleResultSetTable` peut être utilisée directement dans une servlet à l'aide du code montré à l'*exemple 16-3*.

Exemple 16-3. Utilisation de SimpleResultSetTable

```
Statement stmt = con.createStatement();
boolean gotResultSet = stmt.execute(sql);  // SQL de l'utilisateur.

if (!gotResultSet) {
```

Exemple 16-3. Utilisation de SimpleResultSetTable (suite)

```
  out.println(stmt.getUpdateCount() + " lignes mises à jour.");
}
else {
  out.println(new SimpleResultSetTable(stmt.getResultSet()));
}
```

Le tableau généré est montré à la *figure 16-1*.

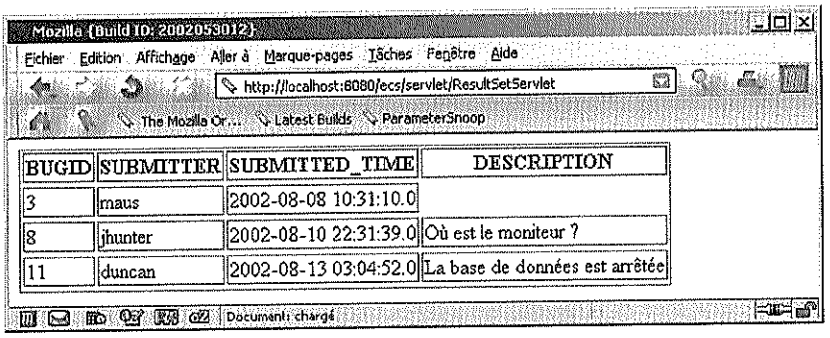


Figure 16-1. Visualisation brute des résultats

Cette classe peut être utilisée conjointement aux technologies de modèles. Il n'y a aucune raison pour que le modèle génère tout le contenu HTML lui-même. Si une partie de la page demande la puissance d'ECS, le modèle peut inclure la sortie d'un composant ECS au bon endroit dans la page.

Que se passe-t-il si le texte renvoyé par le `ResultSet` contient des caractères traités de façon spécial par HTML, comme <, > et & ? Par défaut, ils sont inclus directement et peuvent donc casser la structure du HTML. Pour résoudre ce problème, ECS offre la classe `org.apache.ecs.filter.CharacterFilter` qui convertit les caractères HTML spéciaux en leur entité caractère correspondante. Chaque élément utilise ce filtre par défaut, mais le filtrage est désactivé par défaut afin d'améliorer les performances. Vous pouvez activer le filtrage élément par élément en appelant `setFilterState(true)`, ou vous pouvez l'activer pour le système entier en modifiant *ecs.properties* et en donnant la valeur `true` à `filter_state` et à `filter_attribute_state`. Ensuite, tous les caractères spéciaux présents dans la sortie sont automatiquement convertis en entités caractère.

Personnaliser l'affichage

ECS utilise la nature orientée objet de Java pour fournir un affichage des données hautement personnalisable. Avec un peu de code Java, nous pouvons nous reposer sur `SimpleResultSetTable` et créer une classe `ResultSetTable` moins « simpliste ». Cette nouvelle classe accepte un tableau d'objets `TableCustomizer` qui contrôlent le contenu ajouté au tableau HTML. L'*exemple 16-4* montre l'interface `TableCustomizer`.

Exemple 16-4. L'interface TableCustomizer

```
import java.sql.*;
import org.apache.ecs.*;
import org.apache.ecs.html.*;

public interface TableCustomizer {
  public boolean accept(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException;

  public Element display(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException;
}
```

La méthode accept() doit retourner true si la classe implémentant veut gérer la cellule courante du tableau et sinon false. Pour l'aider dans sa décision, la méthode accept() a accès au type de colonne, au nom du type de colonne, au nom de colonne, à l'objet ResultSet et à l'indice de colonne. La méthode display() est appelée si accept() retourne true ; elle crée et retourne un Element contenant les données de la cellule du tableau. Tous les composants HTML implémentant l'interface Element, en retournant un Element, la méthode display() peut renvoyer n'importe quel type de composant.

Nous pouvons écrire une classe de personnalisation pour mieux afficher les valeurs nulles, les dates et les nombres, comme montré aux *exemples 16-5*, *16-6* et *16-7*. En créant des objets de composant HTML à la place de chaînes de caractères, on obtient un code plus simple et plus extensible. Il serait aussi possible que le tableau limite par programmation le type d'éléments que l'on peut retourner.

Exemple 16-5. Une classe de personnalisation pour les valeurs nulles

```
import java.sql.*;
import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class NullCustomizer implements TableCustomizer {
  public boolean accept(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    rs.getObject(index);
    return rs.wasNull();
  }

  public Element display(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    // Afficher "N/A" pour les entrées nullées.
    return new StringElement("N/A");
  }
}
```

Exemple 16-6. Une classe de personnalisation pour les dates

```
import java.sql.*;
import java.text.*;
import java.util.*;
import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class DateCustomizer implements TableCustomizer {

  DateFormat fmt;

  public DateCustomizer(Locale loc) {
    fmt = DateFormat.getDateTimeInstance(
                    DateFormat.SHORT, DateFormat.SHORT, loc);
  }

  public boolean accept(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    return (columnType == Types.DATE || columnType == Types.TIMESTAMP);
  }

  public Element display(int columnType, String columnTypeName,
                         String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    // Afficher une date et une heure au format court dans la locale indiquée.
    return new StringElement(fmt.format(rs.getDate(index)));
  }
}
```

Exemple 16-7. Une classe de personnalisation pour les nombres

```
import java.sql.*;
import java.text.*;
import java.util.*;
import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class NumberCustomizer implements TableCustomizer {

  NumberFormat fmt;

  public NumberCustomizer(Locale loc) {
    fmt = NumberFormat.getNumberInstance(loc);
  }

  public boolean accept(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    return (columnType == Types.TINYINT ||
```

Exemple 16-7. Une classe de personnalisation pour les nombres (suite)

```
            columnType == Types.SMALLINT ||
            columnType == Types.INTEGER ||
            columnType == Types.BIGINT ||
            columnType == Types.REAL ||
            columnType == Types.FLOAT ||
            columnType == Types.DOUBLE);
  }

  public Element display(int columnType, String columnTypeName,
                         String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    // Afficher le nombre pour la locale indiquée.
    if (columnType == Types.TINYINT ||
        columnType == Types.SMALLINT ||
        columnType == Types.INTEGER ||
        columnType == Types.BIGINT) {
      return new StringElement(fmt.format(rs.getLong(index)));
    }
    else {
      return new StringElement(fmt.format(rs.getDouble(index)));
    }
  }
}
```

Nous pouvons aussi écrire une classe de personnalisation plus avancée afin que tous les identifiants de bogues du tableau soient un lien hypertexte vers une servlet d'affichage du bogue. Ainsi, quelle que soit la requête entrée par l'utilisateur, les identifiants de bogues seront automatiquement créés comme des liens hypertexte. L'*exemple 16-8* montre celle classe de personnalisation.

Exemple 16-8. Une classe de personnalisation pour les bogues

```
import java.sql.*;
import java.text.*;
import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class BugIdCustomizer implements TableCustomizer {

  String bugViewServlet;

  public BugIdCustomizer(String bugViewServlet) {
    this.bugViewServlet = bugViewServlet;
  }

  public boolean accept(int columnType, String columnTypeName,
                        String columnName, ResultSet rs, int index)
                                                  throws SQLException {
    return ((columnType == Types.CHAR ||
```

Exemple 16-8. Une classe de personnalisation pour les bogues (suite)

```
            columnType == Types.VARCHAR ||
            columnType == Types.LONGVARCHAR) &&
           "bugid".equalsIgnoreCase(columnName));
  }

  public Element display(int columnType, String columnTypeName,
                         String columnName, ResultSet rs, int index)
                                                    throws SQLException {
    // Créer un lien vers une servlet d'affichage de ce bogue.
    String bugid = rs.getString(index);
    return new A(bugViewServlet + "?bugid=" + bugid, bugid);
  }
}
```

Le constructeur de la classe `ResultSetTable` accepte un tableau d'objets `TableCustomizer`. Les classes de personnalisation ont l'opportunité de contrôler la création de chaque cellule du tableau. Les classes sont invoquées dans l'ordre de placement dans le tableau et la première à accepter la cellule du tableau est la gagnante. Le code de la classe `ResultSetTable` est donné à l'*exemple 16-9*.

Exemple 16-9. Un tableau de résultats personnalisable

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.ecs.*;
import org.apache.ecs.html.*;

public class ResultSetTable extends Table {

  public ResultSetTable(ResultSet rs) throws SQLException {
    this(rs, null);
  }

  public ResultSetTable(ResultSet rs, TableCustomizer[] customizers)
                                        throws SQLException {
    setBorder(1);

    if (customizers == null) {
      customizers = new TableCustomizer[0];
    }

    ResultSetMetaData rsmd = rs.getMetaData();
    int colCount = rsmd.getColumnCount();

    TR row = new TR();
```

Exemple 16-9. Un tableau de résultats personnalisable (suite)

```
    for (int i = 1; i <= colCount; i++) {
      addElement(new TH().addElement(rsmd.getColumnName(i)));
    }
    addElement(row);

    while (rs.next()) {
      row = new TR();
      for (int i = 1; i <= colCount; i++) {
        TD td = new TD();
        int columnType = rsmd.getColumnType(i);
        String columnTypeName = rsmd.getColumnTypeName(i);
        String columnName = rsmd.getColumnName(i);

        // Donner à chaque classe de personnalisation l'opportunité de
        // contrôler la sortie.
        boolean customized = false;
        for (int c = 0; c < customizers.length; c++) {
          TableCustomizer customizer = customizers[c];
          if (customizer.accept(columnType, columnTypeName,
                                columnName, rs, i)) {
            td.addElement(customizer.display(columnType, columnTypeName,
                                             columnName, rs, i));
            customized = true;
            break;
          }
        }

        // Si aucune classe ne veut du travail, afficher la valeur comme un
        // String
        if (!customized) {
          td.addElement(rs.getString(i));
        }

        addElement(td);
      }
      addElement(row);
    }
  }
}
```

La boucle `while` externe itère sur les lignes du tableau, la boucle `for` externe itère sur les colonnes du tableau, et la boucle `for` interne gère la nouvelle logique des classes de personnalisation. La première classe qui accepte la cellule prend en charge son affichage. Si aucune classe n'accepte la cellule, le tableau affiche une simple valeur `String`. La servlet appelant `ResultSetTable` est montrée à l'*exemple 16-10*

.

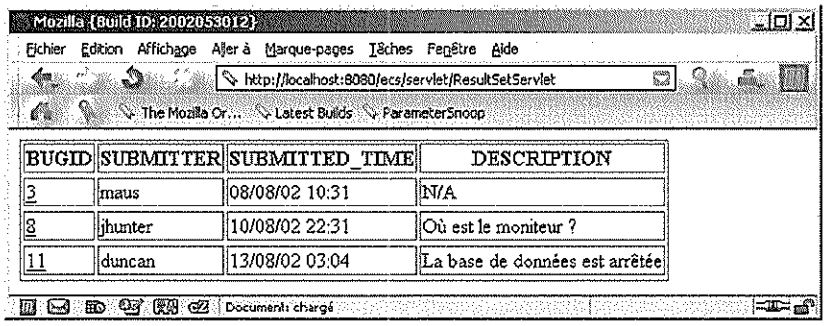
Exemple 16-10. Une servlet testant ResultSetTable

```
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.ecs.*;
import org.apache.ecs.html.*;

import com.oreilly.servlet.*;

public class ResultSetServlet extends HttpServlet {

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    String url = req.getParameter("url");
    String driver = req.getParameter("driver");
    String sql = req.getParameter("sql");

    // Vérifier rapidement que l'url, le pilote et la requête existent.
    ParameterParser parser = new ParameterParser(req);
    String[] required = { "url", "driver", "sql" };
    String[] missing = parser.getMissingParameters(required);
    if (missing != null && missing.length > 0) {
      res.sendError(res.SC_BAD_REQUEST,
                    "L'URL, le pilote et la chaîne SQL doivent être donnés");
      return;
    }

    String param1 = req.getParameter("param1");
    String param2 = req.getParameter("param2");
    String param3 = req.getParameter("param3");
    String param4 = req.getParameter("param4");
    String param5 = req.getParameter("param5");
    String param6 = req.getParameter("param6");
    String val1 = req.getParameter("val1");
    String val2 = req.getParameter("val2");
    String val3 = req.getParameter("val3");
    String val4 = req.getParameter("val4");
    String val5 = req.getParameter("val5");
    String val6 = req.getParameter("val6");

    Properties props = new Properties();
    if (param1 != null && val1 != null) { props.put(param1, val1); }
```

Exemple 16-10. Une servlet testant ResultSetTable (suite)

```
    if (param2 != null && val2 != null) { props.put(param2, val2); }
    if (param3 != null && val3 != null) { props.put(param3, val3); }
    if (param4 != null && val4 != null) { props.put(param4, val4); }
    if (param5 != null && val5 != null) { props.put(param5, val5); }
    if (param6 != null && val6 != null) { props.put(param6, val6); }

    Connection con = null;

    try {
      Class.forName(driver);
      con = DriverManager.getConnection(url, props);

      Statement stmt = con.createStatement();
      boolean gotResultSet = stmt.execute(sql);

      if (!gotResultSet) {
        out.println(stmt.getUpdateCount() + " lignes mises à jour.");
      } else {
        TableCustomizer[] customizers = {
          new NullCustomizer(),
          new DateCustomizer(req.getLocale()),
          new BugIdCustomizer(req.getContextPath() + "/servlet/BugView"),
          new NumberCustomizer(req.getLocale()),
        };
        out.println(new ResultSetTable(stmt.getResultSet(), customizers));
      }
    } catch (Exception e) {
      throw new ServletException(e);
    }
  }
}
```

La *figure 16-2* montre un exemple de sortie. Notez comment les identifiants de bogues sont des liens hypertexte, comment la date est proprement mise en forme et comment la description nulle s'affiche « N/A ».



BUGID	SUBMITTER	SUBMITTED_TIME	DESCRIPTION
3	maus	08/08/02 10:31	N/A
8	jhunter	10/08/02 22:31	Où est le moniteur ?
11	duncan	13/08/02 03:04	La base de données est arrêtée

Figure 16-2. Un affichage personnalisé des résultats (dates européennes)

L'*exemple 16-11* donne une interface HTML simple à la servlet.

Exemple 16-11. Une interface HTML simple à ResultSetServlet

```
<HTML><HEAD><TITLE>SQL Query</TITLE></HEAD><BODY>

<P>
  Cette application exécute une requête SQL ou une mise à jour
  de la base de données via l'Internet public.
</P>

<FORM METHOD="POST" ACTION="servlet/ResultSetServlet">
  <TABLE WIDTH="75%" BORDER="1">
    <TR>
      <TD WIDTH="35%">URL de la base de données :</TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="url" SIZE="60"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%">Pilote de la base de données :</TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="driver" SIZE="60"></TD>
    </TR>
    <TR>
      <TD COLSPAN="2">
        <P ALIGN=CENTER> -- Propriétés de la base de données -- </p>
      </TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop1" VALUE="username"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val1"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop2" VALUE="password"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val2"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop3" VALUE="cacherows"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val3"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop4"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val4"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop5"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val5"></TD>
    </TR>
    <TR>
      <TD WIDTH="35%"><INPUT TYPE=TEXT NAME="prop6"></TD>
      <TD WIDTH="65%"><INPUT TYPE=TEXT NAME="val6"></TD>
    </TR>
```

Exemple 16-11. Une interface HTML simple à ResultSetServlet (suite)

```
  </TABLE>
  <P>
    <TEXTAREA NAME="sql" COLS="80" ROWS="5"></TEXTAREA>
    <BR>
    <INPUT TYPE=SUBMIT VALUE="Soumettre la requête">
  </P>
</FORM>
```

Cette présentation d'ECS n'a fait qu'égratigner la surface de ce qu'il est possible de réaliser. Avec toute la puissance de Java à votre disposition, vous pouvez créer des pages ou des parties de page qui s'adaptent parfaitement à vos données. En exploitant la nature orientée objet d'ECS, vous pouvez vous servir de l'héritage, du polymorphisme et du contrôle de type pour faciliter votre travail.

17

XMLC

XMLC utilise une technologie XML pour que les fichiers HTML deviennent des ressources pour Java. Il a été créé dans le cadre de leur projet en *open source* Enhydra Application Server et on peut le télécharger séparément afin de l'employer dans un autre conteneur de servlets.[1] Dans ce chapitre, nous présentons XMLC 1.2b1, disponible en téléchargement à *http://xmlc.enhydra.org*.[2] XMLC a besoin du JDK 1.1, ou ultérieur, et de n'importe quelle version de l'API Servlet. Sa licence est la Enhydra Public License (EPL), une licence en *open source* similaire à la Mozilla Public License (MPL). Globalement, elle impose que les modifications du code existant soient publiées, mais elle permet aussi les extensions privées, même si elles sont commerciales.

XMLC signifie *XML Compiler*. L'outil prend un document HTML ou XML standard et le « compile » en une classe Java. Cette classe contient les instructions Java nécessaires pour la création en mémoire d'une représentation arborescente XML DOM (*Document Object Model*) du document. Un programmeur peut manipuler l'arbre en mémoire afin d'y ajouter du contenu dynamique et ensuite sortir l'arbre modifié en HTML, XHTML ou autre XML.

Dans une utilisation type, un concepteur de page crée une ou plusieurs maquettes de la page web à développer. Ce sont des fichiers HTML purs. Elles sont présentées aux parties intéressées, discutées, débattues et affinées si nécessaire. Le concepteur n'ajoute pas d'instructions XMLC au fichier, seulement des attributs `ID`, compatibles avec HTML 4.0, dans les zones de la page devant être remplacées par du contenu dynamique.

L'outil XMLC compile la maquette en une classe JAVA contenant la représentation arborescente DOM du document. Le programmeur Java utilise l'API publique de cette classe pour écrire une *classe de manipulation* qui crée une instance de l'arbre, localise les portions du document qui doivent être changées et modifie chaque portion comme né-

1. Enhydra Lutris est le nom scientifique d'une loutre marine, une identité commerciale à propos pour une société basée à Santa Cruz où des loutres marines nagent près de Monterey Bay.

2. *N.d.T.* : au moment de la traduction de ce livre, la version courante d'ECS est la version 2.1. Elle est compatible avec les JDK 1.3 et 1.4.

cessaire. Les parties à modifier sont trouvées à l'aide des méthodes d'accès ajoutées à l'arbre par XMLC. Il en existe pour chaque attribut ID du document. Les fichiers HTML deviennent ainsi des ressources pour Java.

Si la page doit contenir une liste d'élément, la maquette peut en inclure une dizaine afin de paraître réaliste. Le programmeur Java peut supprimer tous les éléments sauf le premier, puis construire la liste réelle en clonant le premier élément ; le style de la maquette est ainsi respecté mais les données sont générées dynamiquement. Il est même possible de demander au compilateur XMLC de supprimer automatiquement tous les éléments excepté le premier, améliorant ainsi les performances à l'exécution.

XMLC obtient un haut niveau de séparation entre le contenu et la présentation. Les fichiers HTML sont en HTML pur, les fichiers Java sont en Java pur et ils se connectent uniquement par des marqueurs ID identifiés. Quand de nouvelles maquettes sont élaborées, ces fichiers peuvent être utilisés directement sans mise à jour et si une balise utilisée est absente de la nouvelle maquette, une erreur de compilation est engendrée lorsque la classe de manipulation invoque la méthode permettant de retrouver l'ID qui n'existe plus.

Une compilation XML simple

Pour apprendre à utiliser XMLC, nous commencerons pas une application de substitution simple. L'*exemple 17-1* montre une maquette de page web simple, *hello.html*, qui accueille l'utilisateur par son nom et lui indique combien de messages l'attendent. Les balises <SPAN>, compatibles avec HTML 4.0, qui entourent le texte sont à remplacer.

Exemple 17-1. Une substitution XMLC simple

```
<!-- hello.html -->
<HTML>
<HEAD><TITLE>Bonjour</TITLE></HEAD>
<BODY>

<H2><SPAN ID="Greeting">Bonjour, Kathlyn</SPAN></H2>
Vous avez <SPAN ID="Messages">103</SPAN> nouveaux messages.

</BODY>
</HTML>
```

Pour exécuter XMLC sur ce fichier, vous devez tout d'abord installer XMLC. Téléchargez l'outil à partir de *http://xmlc.enhydra.org*, extrayez l'archive et suivez les instructions données dans *README* pour l'exécution de *xmlc-config*. Vous êtes ensuite prêt à lancer le compilateur *xmlc* :

```
xmlc -class Hello -keep hello.html
```

Cette commande indique à XMLC de compiler le fichier *hello.html* en une classe nommée Hello ; nous pouvons ensuite examiner le fichier *Hello.java*. Les fichiers HTML standard n'ayant pas besoin d'être du XML bien formé, le compilateur XMLC utilise l'analyseur syntaxique Tidy pour effectuer la conversion de HTML en DOM. Attention cependant, Tidy peut être perturbé par les fichiers HTML qui s'éloignent trop d'un HTML valide. Si cela se produit, vous verrez une longue liste d'avertissements et d'er-

reurs pendant la compilation. Pour éviter ce problème, vous pouvez vous servir d'outils de conception HTML (comme Dreamweaver) pour créer le HTML destiné à XMLC. Pour en savoir plus sur Tidy, rendez-vous sur le site *http://www.w3.org/People/Raggett/tidy*, et sur son portage Java appelé JTidy à *http://www3.sympatico.ca/ac.quick/jtidy.html*.

XMLC accepte une dizaine d'options de compilation. Les plus couramment utilisées sont donnés ci-dessous.[3] Voici la syntaxe de la ligne de commande :

```
xmlc [options] docfile
```

`-class Classname`

Spécifie le nom complet de la classe générée. Par exemple, `Hello` ou `com.servlets.xmlc.Hello`.

`-keep`

Demande de conserver le source Java généré. Très pratique quand on débute avec XMLC.

`-methods`

Affiche la signature de chaque méthode d'accès générée. Aussi très pratique quand on débute.

`-verbose`

Génère une sortie supplémentaire à propos du processus de compilation.

`-urlmapping` *ancienneURL nouvelleURL*

Spécifie une règle de remplacement d'URL. Toutes les instances de *ancienneURL* deviennent *nouvelleURL*. Cela permet de modifier des URL entre les phases de maquettage et de production. Cette option peut être donnée plusieurs fois.

`-urlregexpmapping` *regexp subst*

Spécifie une règle de remplacement d'URL avancée, à l'aide d'expressions régulières. Cette option peut être donnée plusieurs fois.

`-urlsetting` *id nouvelleURL*

Spécifie, par son identifiant de balise, une URL à modifier dans l'arbre DOM. L'URL dont l'identifiant est *id* sera changée en *nouvelleURL*.

`-javac` *prog*

Spécifie le compilateur Java à utiliser.

`-d` *répertoire*

Spécifie le répertoire de sortie. Il est passé directement au compilateur Java pendant la phase de compilation.

`-delete-class` *nom-de-classe*

Détruire tous les éléments du document dont la valeur de l'attribut `CLASS` est *nom-de-classe*. C'est très utile pour supprimer des données de la maquette, comme nous le verrons plus loin. Cette option peut être donnée plusieurs fois.

---

3. XMLC est aussi disponible comme tâche Ant, pour ceux qui utilise l'excellent outil de construction Ant. Voir *http://jakarta.apache.org*.

`-version`

Affiche la version de XMLC.

Lorsque vous exécutez XMLC avec l'option -keep, vous pouvez examiner le source généré. Celui de *Hello.java* est montré à l'*exemple 17-2*.[4]

Exemple 17-2. Le source généré de Hello.java

```
/*
 ****************************************
 * CODE GÉNÉRÉ PAR XMLC, NE PAS MODIFIER *
 ****************************************
 */
import org.w3c.dom.*;
import org.enhydra.xml.xmlc.XMLCUtil;
import org.enhydra.xml.xmlc.XMLCError;
import org.enhydra.xml.xmlc.dom.XMLCDomFactory;

public class Hello extends org.enhydra.xml.xmlc.html.HTMLObjectImpl {
    /**
     * Champ utilisé pour indiquer qu'il s'agit d'une classe générée
     * par XMLC. Contient une référence à l'objet de classe.
     */
    public static final Class XMLC_GENERATED_CLASS = Hello.class;

    /**
     * Champ contenant le nom du fichier source relatif au CLASSPATH
     * à partir duquel cette classe a été générée.
     */
    public static final String XMLC_SOURCE_FILE = "hello.html";

    /**
     * Obtenir l'élément d'identifiant <CODE>Greeting</CODE>.
     * @see org.w3c.dom.html.HTMLElement
     */
    public org.w3c.dom.html.HTMLElement getElementGreeting() {
        return $elementGreeting;
    }
    private org.w3c.dom.html.HTMLElement $elementGreeting;

    /**
     * Fixer la valeur de type texte de l'élément
     * <CODE>Greeting</CODE>.
     * @see org.w3c.dom.Text
     */
    public void setTextGreeting(String text) {
        XMLCUtil.getFirstText($elementGreeting).setData(text);
```

4. *N.d.T.* : le source généré par XMLC contient des commentaires en anglais. Ils ont été traduits ici, malgré l'avertissement donné au début du fichier source.

Exemple 17-2. Le source généré de Hello.java (suite)

```
    }

    /**
     * Obtenir l'élément d'identifiant <CODE>Messages</CODE>.
     * @see org.w3c.dom.html.HTMLElement
     */
    public org.w3c.dom.html.HTMLElement getElementMessages() {
        return $elementMessages;
    }
    private org.w3c.dom.html.HTMLElement $elementMessages;

    /**
     * Fixer la valeur de type texte de l'élément
     * <CODE>Messages</CODE>.
     * @see org.w3c.dom.Text
     */
    public void setTextMessages(String text) {
        XMLCUtil.getFirstText($elementMessages).setData(text);
    }

    /**
     * Créer un objet document.
     */
    private static Document createDocument() {
        XMLCDomFactory domFactory =
            new org.enhydra.xml.xmlc.dom.DefaultHTMLDomFactory();
        Document document = domFactory.createDocument(null, null);
        return document;
    }

    /**
     * Créer un document DOM et initialiser les champs des méthodes d'accès.
     */
    public void buildDocument() {
        Document document = createDocument();
        setDocument(document);
        Node $node0, $node1, $node2, $node3, $node4;
        Element $elem0, $elem1, $elem2, $elem3;
        Attr $attr0, $attr1, $attr2, $attr3;

        $elem0 = document.getDocumentElement();
        $elem1 = document.createElement("HEAD");;
        $elem0.appendChild($elem1);

        $elem2 = document.createElement("TITLE");;
        $elem1.appendChild($elem2);

        $node3 = document.createTextNode("Hello");;
        $elem2.appendChild($node3);
```

Exemple 17-2. Le source généré de Hello.java (suite)

```
        $elem1 = document.createElement("BODY");;
        $elem0.appendChild($elem1);

        $elem2 = document.createElement("H2");;
        $elem1.appendChild($elem2);

        $elem3 = document.createElement("SPAN");;
        $elem2.appendChild($elem3);

        $elementGreeting = (org.w3c.dom.html.HTMLElement)$elem3;
        $attr3 = document.createAttribute("id");
        $attr3.setValue("Greeting");
        $elem3.setAttributeNode($attr3);

        $node4 = document.createTextNode("Bonjour, Kathlyn");;
        $elem3.appendChild($node4);

        $node2 = document.createTextNode("Vous avez ");;
        $elem1.appendChild($node2);

        $elem2 = document.createElement("SPAN");;
        $elem1.appendChild($elem2);

        $elementMessages = (org.w3c.dom.html.HTMLElement)$elem2;
        $attr2 = document.createAttribute("id");
        $attr2.setValue("Messages");
        $elem2.setAttributeNode($attr2);

        $node3 = document.createTextNode("103");;
        $elem2.appendChild($node3);

        $node2 = document.createTextNode(" nouveaux messages.");;
        $elem1.appendChild($node2);

        $node1 = document.createComment(" hello.html ");;
        $elem0.appendChild($node1);
    }

    /**
     * Fonction récursive pour définir les champs des méthodes d'accès à
     * partir du DOM. Les identifiants manquants sont mis à null.
     */
    protected void syncWithDocument(Node node) {
        if (node instanceof Element) {
            String id = ((Element)node).getAttribute("id");
            if (id.length() == 0) {
            } else if (id.equals("Greeting")) {
                $elementGreeting = (org.w3c.dom.html.HTMLElement)node;
            } else if (id.equals("Messages")) {
```

Exemple 17-2. Le source généré de Hello.java (suite)

```
                $elementMessages = (org.w3c.dom.html.HTMLElement)node;
            }
        }
    }

    /**
     * Constructeur par défaut.
     */
    public Hello() {
        buildDocument();
    }

    /**
     * Constructeur avec constuction optionnelle du DOM.
     *
     * @param buildDOM Si false, le DOM ne sera pas construit tant que
     * buildDocument() n'est pas appelée par la classe dérivée. Si true,
     * le DOM est construit immédiatement.
     */
    public Hello(boolean buildDOM) {
        if (buildDOM) {
            buildDocument();
        }
    }

    /**
     * Constructeur de copie.
     * @param src Le document à cloner.
     */
    public Hello(Hello src) {
        setDocument((Document)src.getDocument().cloneNode(true));
        syncAccessMethods();
    }

    /**
      * Cloner le document.
      * @param deep Doit valoir true, car seul le clonage en profondeur est
      * supporté.
      */
    public Node cloneNode(boolean deep) {
        cloneDeepCheck(deep);
        return new Hello(this);
    }
}
```

Cet exemple contient beaucoup de code, mais il est assez simple. La classe se nomme `Hello`, comme nous l'avons demandé par le biais de l'option `-class`. Elle étend `org.enhydra.xml.xmlc.html.HTMLObjectImpl`, la classe mère XMLC pour tous les objets HTML, et elle implémente `org.w3c.dom.html.HTMLDocument`, l'interface DOM stan-

dard représentant un document HTML. Le constructeur `Hello` appelle la méthode `buildDocument()` pour remplir l'arbre DOM avec le contenu provenant du fichier HTML. DOM n'est qu'un ensemble d'interfaces et il est possible de choisir parmi plusieurs implémentations de DOM. Nous pouvons voir dans la méthode `createDocument()` que cette classe utilise l'implémentation XMLC par défaut renvoyée par `DefaultHTMLDomFactory`, qui, dans la version 1.2b1, est Apache Xerces (voir à *http://xml.apache.org*).

Si vous exécutez *xmlc* avec l'option `-methods`, vous pouvez voir un résumé des méthodes d'accès dans le source :

```
% xmlc -class Hello -keep -methods hello.html
public org.w3c.dom.html.HTMLElement getElementGreeting();
public void setTextGreeting(String text);
public org.w3c.dom.html.HTMLElement getElementMessages();
public void setTextMessages(String text);
```

Les méthodes `get` retournent l'objet `HTMLElement` de DOM représentant l'élément HTML marqué avec l'attribut `ID`. Les méthodes `set` fixent le contenu textuel de ces éléments. Notez que la valeur de l'attribut `ID` est codée dans le nom des méthodes. Les valeurs qui ne peuvent être converties en des identifiants Java valides n'auront pas de méthodes d'accès ; soyez donc prudents et n'utilisez que des lettres, des chiffres et des caractères souligné (_). Rappelez-vous aussi que les valeurs d'identifiants sont supposées uniques dans la portée de la page.

La classe de manipulation

Une fois le fichier HTML compilé en une représentation XML basée sur Java, l'étape suivante est d'écrire ce que l'on appelle une *classe de manipulation*. Cette classe crée une instance du document, modifie son contenu et envoie le document vers sa destination finale. L'*exemple 17-3* montre une classe de manipulation simple pour notre page `Hello`. Notez qu'il s'agit d'un programme autonome. Cette classe de manipulation est le plus souvent une servlet ou une classe appelée par une servlet, mais ce n'est pas une obligation. XMLC ne dépend pas de l'API Servlet.

Exemple 17-3. La classe de manipulation de la page Hello

```
import java.io.*;
import org.w3c.dom.*;
import org.w3c.dom.html.*;

import org.enhydra.xml.io.DOMFormatter;

public class HelloManipulation {

  public static void main(String[] args) {
    // Du contenu pseudo-dynamique.
    String username = "Mark Diekhans";
    int numMessages = 43;
```

Exemple 17-3. La classe de manipulation de la page Hello (suite)

```
    // Créer l'arbre DOM.
    Hello hello = new Hello();

    // Définir le titre, à l'aide d'une méthode DOM standard.
    hello.setTitle("Bonjour XMLC !");

    // Définir la valeur de "greeting".
    hello.setTextGreeting("Bonjour, " + username);

    // Définir la valeur de "messages"
    hello.setTextMessages("" + numMessages);

    try {
      DOMFormatter formatter = new DOMFormatter(); // Peut être beaucoup
                                                   // ajusté.
      formatter.write(hello, System.out);
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe définit d'abord les valeurs pseudo-dynamiques à ajouter à la page : le nom d'utilisateur et le nombre de messages. Puis, elle crée une instance de la classe `Hello` où se trouve le contenu du document HTML. Si l'instance `hello` était sortie à ce stade, elle générerait une représentation fidèle du fichier HTML originale. La classe de manipulation définit alors le titre de la page, le contenu du texte d'accueil et le nombre de messages. Le texte d'accueil et le nombre de messages sont fixés à l'aide des méthodes d'accès ajoutées par XMLC. Le titre est défini par le biais d'une méthode DOM standard. Il existe plusieurs méthodes d'accès standard intéressantes dans les classes DOM HTML, par exemple `getApplets()`, `getImages()`, `getLinks()`, `getForms()`, `getAnchors()` et ainsi de suite. Pour plus d'informations sur les possibilités des classes DOM, reportez-vous à la documentation fournie par XMLC. Enfin, la classe de manipulation utilise un `org.enhydra.xml.io.DOMFormatter` pour écrire l'instance de `hello` sur `System.out`. Le `DOMFormatter` peut accepter une classe `OutputOptions` dans son constructeur afin d'ajuster la sortie du XML. Mais pour les navigateurs web standard, le comportement par défaut est correct et il produit :

```
<HTML><HEAD><TITLE>Bonjour XMLC !</TITLE></HEAD><BODY><H2><SPAN
id='Greeting'>Bonjour, Mark Diekhans</SPAN></H2>Vous avez <SPAN
id='Messages'>43</SPAN> nouveaux messages.
</BODY><!-- hello.html --></HTML>
```

Notez que les valeurs dans la maquette ont été remplacées par de nouvelles valeurs et tous les espaces supplémentaires ont été supprimés pour plus d'efficacité. Notez aussi que le commentaire a été placé à la fin du fichier. C'est un bogue de Xerces.

Pour exécuter vous même la classe de manipulation, vous devez vous assurer que le fichier *xmlc.jar* contenant les classes XMLC se trouve dans votre chemin de classes (ou ce-

lui de votre serveur si l'appel à XMLC se fait à partir d'une servlet). Sachez que *xmlc.jar* contient plusieurs paquetages utilitaires pouvant provoquer des conflits si vous avez installé différentes versions de ces mêmes paquetages. Avec XMLC 1.2b1, faites attention aux collisions externes avec Apache Xerces 1.0.2, SAX 1.0, DOM Level 2, GNU RegExp 1.0.8 et JTidy 26jul1999. Les collisions de paquetage peuvent donner des messages d'erreur mystérieux. Par exemple, lors de son exécution, le compilateur *xmlc* peut indiquer que la classe générée est abstraite et qu'elle ne peut être instanciée parce que la classe ne définit pas une méthode particulière d'une classe particulière. Le véritable problème est : vous avez une incompatibilité entre des versions de DOM. La solution consiste à vous assurer que la dernière version vient en premier dans le chemin de classes. Cela fonctionne tant que les versions des paquetages sont compatibles.

Modifier une liste

Passons maintenant à une opération plus évoluée, modifier une liste. Tout d'abord, nous créons la page prototype contenant une maquette de liste, comme montré à l'*exemple 17-4*.

Exemple 17-4. Une maquette de liste de langues

```
<!-- snoop.html -->
<HTML><HEAD><TITLE>Analyse XMLC</TITLE></HEAD>

<H1>Locales du client</H1>

<UL>
<LI ID="locale">fr
<LI CLASS="mockup">en
<LI CLASS="mockup">es
<LI CLASS="mockup">jp
</UL>

</BODY></HTML>
```

Dans la version en production, cette liste ferait partie d'une page plus grande, mais ce n'est pas du tout important pour XMLC. Nous compilons la page à l'aide du compilateur XMLC et nous spécifions `-delete-class mockup` pour que les éléments marqués `mockup` soient automatiquement supprimés pendant la phase de compilation :

```
% xmlc -class Snoop -keep -delete-class mockup -methods snoop.html
public org.w3c.dom.html.HTMLLIElement getElementLocale();
```

Notez qu'une seule méthode d'accès a été ajoutée au document : `getElementLocale()` retourne l'élément `<ID>` marqué `locale` comme un objet `HTMLLIElement`. La classe de manipulation modifiant dynamiquement ce document est donnée à l'*exemple 17-5*.

Exemple 17-5. La classe de manipulation de la liste de langues

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
```

Exemple 17-5. La classe de manipulation de la liste de langues (suite)

```
import javax.servlet.http.*;

import org.w3c.dom.*;
import org.w3c.dom.html.*;

import org.enhydra.xml.io.DOMFormatter;

public class SnoopManipulation extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // Obtenir les données dynamiques à afficher.
    Enumeration locales = req.getLocales();

    // Créer l'arbre DOM.
    Snoop snoop = new Snoop();

    // Obtenir le premier élément de la liste "prototype".
    // Le reste a été enlevé pendant la compilation par xmlc.
    HTMLLIElement item = snoop.getElementLocale();

    // Obtenir le parent du prototype afin de gérer les fils.
    Node parent = item.getParentNode();

    // Boucler sur les locales, ajoutant un noeud pour chacun.
    while (locales.hasMoreElements()) {
      Locale loc = (Locale)locales.nextElement();
      HTMLLIElement newItem = (HTMLLIElement) item.cloneNode(true);
      Text text = snoop.createTextNode(loc.toString());
      newItem.replaceChild(text, newItem.getLastChild());
      parent.insertBefore(newItem, null);
    }

    // Supprimer l'élément prototype.
    parent.removeChild(item);

    // Sortir le document.
    DOMFormatter formatter = new DOMFormatter(); // Peut être beaucoup ajusté.
    formatter.write(snoop, out);
  }
}
```

La partie intéressante de cette servlet se trouve dans la boucle `while`. Pour chaque locale, la servlet clone l'élément prototype, change le texte contenu par le clone et ajoute le clone à la fin de la liste. Après la fin de la boucle `while`, la servlet supprime l'élément

prototype, obtenant une liste remplie de clones contenant les données réelles. Le résultat est montré à la *figure 17-1*.[5]

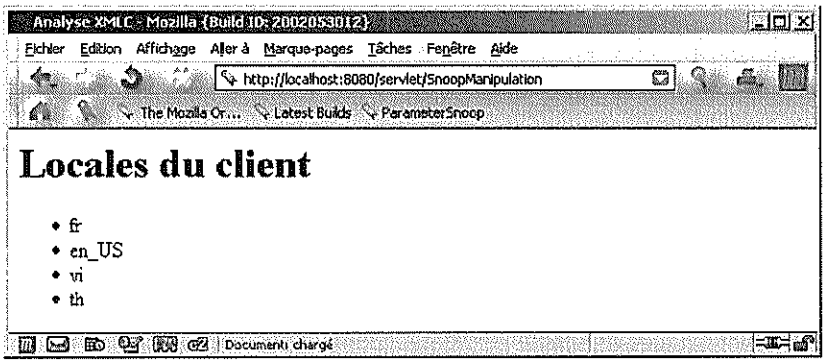


Figure 17-1. Abréviations des locales pour le français, l'anglais (U.S.), le vietnamien et le thaï.

Avec cet exemple, nous pouvons voir que l'une des difficultés de XMLC est sa gestion du modèle objet DOM pas vraiment intuitif. Il faut quelques astuces pour utiliser DOM. Par exemple, vous pouvez voir ici qu'un contenu de type `String` ne peut être créé qu'à l'aide de la méthode de fabrique `createTextNode()` appelée sur le document auquel le texte doit être attaché. Une fois le contenu créé, pour ajouter le texte au document, il faut ajouter le texte sous forme d'un nœud enfant de celui contenant l'élément et supprimer l'enfant existant. Dans une future version de XMLC, il sera sans doute possible d'utiliser le modèle objet de JDOM qui est bien plus facile. Il y a aussi des boîtes à outils en cours de développement apportant une couche XMLC au-dessus de DOM éliminant certaines de ses contraintes les plus pénibles.

Une application outil

Pour conclure notre présentation de XMLC, voyons comment l'application outil utilisée pour illustrer Tea et WebMacro peut être réécrite avec XMLC. Nous verrons tout d'abord le fichier HTML modèle qui donne l'aspect général de notre site mais qui n'inclut aucun contenu relatif à la liste des outils. Il est montré à l'*exemple 17-6*.

Exemple 17-6. Le fichier modèle de l'application outil

```
<!-- template.html -->
<HTML><HEAD><TITLE>Exemple de titre</TITLE></HEAD>

<BODY BGCOLOR="#FFFFFF" BACKGROUND="/images/background.gif"
      LINK="#003333" ALINK="#669999" VLINK="#333333">
```

5. Attention aux interactions entre les chemins de classes quand vous exécutez cet exemple. Il y a des bibliothèques XML nécessaires à XMLC, à JDOM et peut-être au serveur lui-même. Satisfaire les contraintes de version de la bibliothèque XML pour ces trois composants sans créer de conflit peut demander un peu de magie noire et ce n'est pas toujours possible.

Exemple 17-6. Le fichier modèle de l'application outil (suite)

```
<IMG SRC="/images/banner.gif" WIDTH=600 HEIGHT=87 BORDER=0><BR>

<TABLE>
<TR>
<TD WIDTH=125 VALIGN=TOP>
 <BR><BR><BR>
 <FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#FF0000">
 <A HREF="/index.html">Accueil</A><BR>
 <A HREF="/hosting.html">Hébergement</A><BR>
 <A HREF="/engines.html">Moteurs</A><BR>
 </FONT>
</TD>

<TD WIDTH=475>

  <TABLE CELLPADDING=5><TR><TD WIDTH=600 BGCOLOR="#006699" VALIGN=TOP>
  <B><FONT FACE="Arial,Helvetica" SIZE="+2">
  <SPAN ID="title">Exemple de titre</SPAN>
  </FONT></B>
  </TD></TR></TABLE>

  <B><FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#003366">
  <SPAN ID="deck">Exemple de sujet</SPAN>
  </FONT></B><P>

  <P>

  <FONT FACE="Arial,Helvetica">

  <SPAN ID="desc">Exemple de description</SPAN>

  <DIV ID="content">Mettre le vrai contenu ici</DIV>

  </FONT>

</TD>
</TR>

<TR>
<TD></TD>
<TD WIDTH=475 ALIGN=CENTER COLSPAN=3>
<HR>
<FONT FACE="Arial,Helvetica">
<A HREF="/index.html">Accueil</A>  
<A HREF="/hosting.html">Hébergement</A>  
<A HREF="/engines.html">Moteurs</A>  <P>
</FONT>
```

Exemple 17-6. Le fichier modèle de l'application outil (suite)

```
<TABLE WIDTH=100%>
<TR>
<TD WIDTH=260 ALIGN=LEFT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
<A HREF="/copyright.html">Copyright</A> &copy; 2000 Jason Hunter<BR>
Tous droits réservés.</TD>
<TD WIDTH=5></FONT></TD>
<TD WIDTH=230 ALIGN=RIGHT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
Contact : <A HREF="mailto:webmaster@servlets.com">webmaster@servlets.com</a>
</FONT></TD>
</TR>
</TABLE>

</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Notez que le titre de la page est un emplacement et qu'il y a trois éléments <SPAN>, nommés title, deck et desc, qui contiennent le texte maquette destiné à être remplacé. Il y a aussi un élément <DIV>, appelé content, qui marque l'endroit où doit être affiché le vrai contenu de la page. Le fichier HTML utilisé pour générer le contenu de l'application outil est montré à l'*exemple 17-7*.

Exemple 17-7. Le fichier de contenu de l'application outil

```
<!-- toolview.html -->
<HTML><HEAD><TITLE>Liste des outils</TITLE></HEAD>

<BODY BGCOLOR="#FFFFFF" BACKGROUND="/images/background.gif"
      LINK="#003333" ALINK="#669999" VLINK="#333333">

<IMG SRC="/images/banner.gif" WIDTH=600 HEIGHT=87 BORDER=0><BR>

<TABLE>
<TR>
<TD WIDTH=125 VALIGN=TOP>
 <BR><BR><BR>
 <FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#FF0000">
 <A HREF="/index.html">Accueil</A><BR>
 <A HREF="/hosting.html">Hébergement</A><BR>
 <A HREF="/engines.html">Moteurs</A><BR>
 </FONT>
</TD>

<TD WIDTH=475>
```

Exemple 17-7. Le fichier de contenu de l'application outil (suite)

```
<TABLE CELLPADDING=5><TR><TD WIDTH=600 BGCOLOR="#006699" VALIGN=TOP>
<B><FONT FACE="Arial,Helvetica" SIZE="+2">
<SPAN ID="title">Liste des outils</SPAN>
</FONT></B>
</TD></TR></TABLE>

<B><FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#003366">
<SPAN ID="deck">Une liste des outils de création de contenu</SPAN>
</FONT></B><P>

<P>

<FONT FACE="Arial,Helvetica">

<SPAN ID="desc">
Sans les outils, les gens ne sont rien de plus que des animaux. Et plutôt
faibles, en plus. Voici une liste d'outils de création de contenu basé sur
les servlets que vous pouvez utiliser afin de ne plus être une mauviette.
</SPAN>

<DIV ID="record">
  <HR SIZE=2 ALIGN=LEFT>

  <FONT FACE="Arial,Helvetica">
  <H3>
  <SPAN ID="toolName">Un nom d'outil</SPAN>
  <FONT COLOR=#FF0000><B><SPAN ID="toolStatus">(Nouveau !)</SPAN> </B></FONT>
  </H3>
  <A ID="toolLink" HREF="http://toolhome.com">http://toolhome.com</A><BR>

  <SPAN ID="toolComments">
     Commentaires sur l'outil.
  </SPAN>
  </FONT>
</DIV>

<DIV>
  <HR SIZE=2 ALIGN=LEFT>

  <FONT FACE="Arial,Helvetica">
  <H3>
  Autre nom d'outil
  <FONT COLOR=#FF0000><B> (Mis à jour !) </B></FONT>
  </H3>
  <A HREF="http://toolhome.com">http://toolhome.com</A><BR>
     Commentaires sur cet outil.
  </FONT>
</DIV>
```

Exemple 17-7. Le fichier de contenu de l'application outil (suite)

```
  </FONT>

</TD>
</TR>

<TR>
<TD></TD>
<TD WIDTH=475 ALIGN=CENTER COLSPAN=3>
<HR>
<FONT FACE="Arial,Helvetica">
<A HREF="/index.html">Accueil</A>  
<A HREF="/hosting.html">Hosting</A>  
<A HREF="/engines.html">Moteurs</A>  <P>
</FONT>

<TABLE WIDTH=100%>
<TR>
<TD WIDTH=260 ALIGN=LEFT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
<A HREF="/copyright.html">Copyright</A> &copy; 2000 Jason Hunter<BR>
Tous droits réservés.</TD>
<TD WIDTH=5></FONT></TD>
<TD WIDTH=230 ALIGN=RIGHT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
Contact : <A HREF="mailto:webmaster@servlets.com">webmaster@servlets.com</a>
</FONT></TD>
</TR>
</TABLE>

</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Le fichier ressemble beaucoup au fichier modèle, mais nous avons remplacé les valeurs des emplacements par des valeurs réalistes et rempli la zone de contenu avec des exemples d'enregistrements d'outils. Voici comment nous utiliserons ce fichier : par programmation, nous prendrons ses composants clés (son titre, son sujet, sa description et son enregistrement prototype), nous les copierons et les placerons dans le fichier modèle afin de créer la page finale.

Pourquoi avons-nous besoin de deux fichiers ? Pourquoi ne pourrions-nous pas simplement modifier directement le fichier *toolview.html* ? Oui, ce serait possible, mais nous utilisons le modèle de façon telle qu'une mise à jour future de l'en-tête, de la barre de menu ou du bas de page ne demande que la mise à jour du fichier *template.html* pour que toutes les pages soient concernées. Autrement dit, le modèle dicte la mise en page générale de la page. Le fichier *toolview.html* ne sert que pour sa partie critique.

Nous passons les fichiers HTML au compilateur XMLC à l'aide de commandes similaires aux suivantes. Il y aura des avertissements car le HTML n'est pas du XML valide :

```
% xmlc -class Template -keep -methods template.html
% xmlc -class ToolView -keep -methods toolview.html
```

Puis, nous sommes prêts à écrire la servlet qui tient le rôle de classe de manipulation. Elle est donnée à l'*exemple 17-8*.

Exemple 17-8. La classe de manipulation de l'application outil

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.w3c.dom.*;
import org.w3c.dom.html.*;

import org.enhydra.xml.io.DOMFormatter;

public class ToolViewServlet extends HttpServlet {

  private Tool[] tools;

  public void init() throws ServletException {
    // Charger les données sur les outils dans l'initialisation pour plus
    // de simplicité.
    String toolsFile = getInitParameter("toolsFile"); // Vient de web.xml.
    if (toolsFile == null) {
      throw new ServletException(
        "Un fichier de données sur les outils doit être spécifié dans " +
        "le paramètre d'initialisation toolsFile");
    }
    log("Chargement des outils à partir de " + toolsFile);
    try {
      tools = Tool.loadTools(toolsFile);
      if (tools.length == 0) {
        log("Aucun outil trouvé dans " + toolsFile);
      }
      else {
        log(tools.length + " outil(s) trouvé(s) dans " + toolsFile);
      }
    }
    catch (Exception e) {
      throw new ServletException(e);
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
```

Exemple 17-8. La classe de manipulation de l'application outil (suite)

```
PrintWriter out = res.getWriter();

// Créer l'arbre DOM pour le document complet.
Template template = new Template();

// Créer l'arbre DOM pour le contenu interne.
ToolView toolview = new ToolView();

// Obtenir l'enregistrement de visualisation du modèle de l'application
// outil.
HTMLDivElement record = toolview.getElementRecord();

// Obtenir une référence au point d'insertion de la liste d'outils.
HTMLDivElement insertionPoint = template.getElementContent();
Node insertionParent = insertionPoint.getParentNode();

// Définir le titre, le sujet et la description du modèle. Extraire
// les données du fichier toolview.html.
String title=((Text)toolview.getElementTitle().getFirstChild()).getData();
String deck=((Text)toolview.getElementDeck().getFirstChild()).getData();
String desc=((Text)toolview.getElementDesc().getFirstChild()).getData();
template.setTitle(title);     // Le titre de la page.
template.setTextTitle(title); // L'élément marqué "title".
template.setTextDeck(deck);   // L'élément marqué "deck".
template.setTextDesc(desc);   // L'élément marqué "desc".

// Ajouter un enregistrement pour chaque outil.
for (int i = 0; i < tools.length; i++) {
  Tool tool = tools[i];

  toolview.setTextToolName(tool.name);
  toolview.setTextToolComments(tool.comments);

  if (tool.isNewWithin(45)) {
    toolview.setTextToolStatus(" (Nouveau !) ");
  }
  else if (tool.isUpdatedWithin(45)) {
    toolview.setTextToolStatus(" (Mis à jour !) ");
  }
  else {
    toolview.setTextToolStatus("");
  }

  HTMLAnchorElement link = toolview.getElementToolLink();
  link.setHref(tool.homeURL);
  Text linkText = toolview.createTextNode(tool.homeURL);
  link.replaceChild(linkText, link.getLastChild());
```

Exemple 17-8. La classe de manipulation de l'application outil (suite)

```
        // importNode() existe dans DOM Niveau 2.
        insertionParent.insertBefore(template.importNode(record, true), null);
      }

      // Supprimer l'emplacement d'insertion.
      insertionParent.removeChild(insertionPoint);

      // Sortir le document.
      DOMFormatter formatter = new DOMFormatter(); // Peut être beaucoup ajusté.
      formatter.write(template, out);
    }
}
```

La méthode `init()` de la servlet charge, à l'aide de la classe `Tool` du chapitre 14, *L'environnement Tea*, les données sur les outils à partir du fichier indiqué dans le paramètre d'initialisation `toolsFile`. C'est dans la méthode `doGet()` que se fait le travail intéressant. Nous créons une instance du document `Template` et une instance du document `ToolView`. Ensuite, nous localisons l'enregistrement de l'outil dans `ToolView` et le point d'insertion des enregistrements d'outil dans `Template`. Puis nous retrouvons le titre, le sujet et la description dans le document `ToolView` et copions ces valeurs dans `Template`.

Dans la boucle `for`, nous créons la liste des enregistrements d'outils. Pour chaque outil, nous attribuons les valeurs appropriées à l'enregistrement prototype : tout d'abord le nom et les commentaires, puis l'état et enfin le lien. Après la modification de l'enregistrement, nous l'ajoutons au modèle au point d'insertion. Dans DOM, les nœuds sont liés au document qui les a créés. Nous employons donc la méthode `importNode()` pour que l'enregistrement passe du document `ToolView` à `Template`. La méthode `importNode()` effectue une copie en profondeur (car son second paramètre vaut `true`) afin que chaque itération de la boucle `for` ajoute une copie différente du contenu de l'enregistrement. Après la boucle `for`, nous supprimons le point d'insertion du modèle et envoyant le document au client.

Lorsque vous exécutez cette servlet, sachez que la méthode `importNode()` est nouvelle dans DOM Level 2. Pour que cette servlet fonctionne, votre serveur doit donc avoir un chemin de classes qui inclut les classes de DOM Level 2 avant celles de DOM Level 1. L'archive *xmlc.jar* contenant DOM Level 2, vous pourriez croire que ce n'est pas un problème, mais certains serveurs (dont Tomcat 3.2) dispose d'une implémentation de DOM Level 1 dans leur chemin de classe par défaut afin de pouvoir lire les fichiers *web.xml*. Pour que cette servlet fonctionne sur de tels serveurs, vous devez modifier le chemin de classes du serveur afin que le fichier *xmlc.jar* de XMLC soit trouvé avant les bibliothèques XML du serveur. (Tomcat 3.2 chargeant automatiquement les fichiers JAR par ordre alphabétique, vous devrez peut-être renommer les fichiers JAR.)

Un problème inhérent à XMLC, dévoilé par cet exemple, concerne l'inclusion conditionnelle de blocs de texte dont le contenu varie, comme les remarques (*Nouveau !*) et (*Mis à jour !*), qui peut se révéler difficile dans des pages dont le modèle ne déclare l'inclusion que d'un seul bloc de texte. Dans cet exemple, il est clair que le code Java en sait plus qu'il ne devrait sur l'affichage des remarques.

Un avantage de XMLC est que les caractères spéciaux dans l'élément inclus, comme <, > et &, sont automatiquement passés à l'échappement lors de l'envoi au client, car le système de formatage de la sortie du document peut facilement distinguer ce qui fait partie de la structure et ce qui est du contenu. Cette connaissance de la structure du document permet aussi à ce système d'effectuer des tâches avancées comme la modification de tous les liens pour coder les identifiants de session, bien que cette fonctionnalité n'a pas encore été implémentée.

La page générée par cette servlet est similaire à celle générée par les exemples de Tea et de WebMacro.

18

JavaServer Pages

Les JavaServer Pages, connues sous le nom JSP, constituent une technologie créée par Sun Microsystems et très étroitement liée aux servlets. Les JSP ont été l'un des premiers efforts de développement d'un système de création de contenus basé sur les servlets non propriétaire. Si vous êtes au contact des servlets depuis longtemps, vous vous souvenez peut-être que les JSP ont été annoncées pour la première fois au printemps 1998 ; en fait la première édition de ce livre n'a pu inclure qu'une courte présentation de la version O.91 des JSP. Bien entendu, les JSP ont beaucoup évolué depuis. Dans ce chapitre, nous examinons les JSP 1.1, construites au-dessus de l'API Servlet 2.2.

Comme avec les servlets, Sun a publié une spécification des JSP (créée par un groupe d'experts constitué de fournisseurs et de personnes extérieures à Sun) et les fournisseurs tiers sont en concurrence sur l'implémentation de cette norme. Contrairement aux autres technologies présentées et construites au-dessus des servlets, les JSP sont une spécification, non un produit, et, pour fonctionner, elles ont besoin du support du serveur. La plupart des fournisseurs de conteneurs de servlets apportent ce support, y compris Tomcat dont le moteur de JSP s'appelle Jasper. Les JSP sont aussi l'un des composants fondamentaux de Java 2, Enterprise Edition (J2EE).

Un objectif avoué des JSP (extrait de la spécification) est de « permettre la séparation du contenu statique et du contenu dynamique ». Un autre but est de « permettre la conception de pages web qui créent facilement du contenu dynamique mais avec un maximum de puissance et de souplesse ». Les JSP remplissent plutôt bien ces deux objectifs. Cependant, du fait de la « puissance maximale » inhérente à sa conception des JSP, l'auteur d'une page JSP a toujours le contrôle du système et l'on peut dire que les JSP *permettent* la séparation du contenu de la présentation mais ne la *garantissent* ni ne l'*encouragent* autant que les autres technologies. Cela ressemble à la manière dont C++ permet une bonne conception orientée objet mais dont la structure ne l'encourage pas autant que Java.

Les JSP sont aussi une technologie très souple et il existe plusieurs manières de les employer. Tout d'abord, les JSP peuvent être un raccourci pour l'écriture de servlets : un programmeur de servlets, à l'aise en Java, peut placer du code Java directement dans une page JSP au lieu d'écrire une servlet complète, obtenant ainsi certains avantages

comme l'élimination des appels à `out.println()`, la possibilité de désigner directement un fichier JSP et exploiter la fonction de compilation automatique offerte par les JSP. Cependant, en écrivant une page JSP à la place d'une servlet, le programmeur perd le contrôle total du code et de l'environnement d'exécution, lui empêchant d'effectuer des choses comme étendre `com.oreilly.servlet.CacheHttpServlet` ou générer une sortie binaire (image). C'est pourquoi, le codage doit rester dans les servlets, les pages JSP servant tout d'abord à la logique de présentation.

Même lorsqu'elles sont employées uniquement pour la présentation, les pages JSP peuvent être utilisées de plusieurs manières. Par exemple, il est possible d'inclure des JavaBeans dans la page. On peut aussi créer des balises personnalisées qui ressemblent à du HTML mais qui en fait permettent de supporter du code Java. Il est aussi possible d'envoyer toutes les requêtes à une servlet qui se charge de la logique métier et de faire en sorte que cette servlet délègue l'affichage de la requête à une page JSP en utilisant le mécanisme `RequestDispatcher`. Cette technique est souvent appelée l'*architecture du modèle 2*, un nom provenant de la spécification des JSP 0.92. Il existe d'autres architectures, certaines ayant des noms fantasques comme *Modèle 1½*, *Modèle 2½* ou *Modèle 2+1*. Il n'y a pas de limites à vos meilleures idées d'utilisation des JSP.

Dans ce chapitre, nous examinerons les différentes façons d'utiliser les JSP, du raccourci pour l'écriture de servlets aux balises personnalisées. Les descriptions seront rapides mais vous aurez les bases vous permettant de comparer les JSP et les autres solutions. Nous ne verrons pas toutes les options architecturales et nous nous limiterons aux aspects techniques. Pour les différentes architectures et pour d'autres informations sur les Java-Server Pages, visitez le site *http://java.sun.com/products/jsp* (une page décrivant la syntaxe des JSP 1.0 est disponible à *http://java.sun.com/products/jsp/syntax.pdf*) et le livre *JavaServer Pages* de Hans Bergsten (Éditions O'Reilly).

Utiliser les JavaServer Pages

Les JSP permettent l'insertion directe de code de servlet dans un fichier HTML qui sans cela resterait statique.[1] Chaque bloc du code de servlet (appelé une *scriptlet*) est entouré par la balise d'ouverture <% et celle de fermeture %>. Une scriptlet peut utiliser un certain nombre de variables prédéfinies. Voici les six variables les plus utilisées :

`HttpServletRequest request`

La requête de la servlet.

`HttpServletResponse response`

La réponse de la servlet.

`javax.servlet.jsp.JspWriter out`

L'écrivain de sortie, utilisé comme un `PrintWriter` mais avec des caractéristiques d'utilisation du tampon différentes.

1. Avant d'aller plus loin, nous devons signaler que placer du code Java dans une page JSP ne constitue pas une bonne façon de procéder. Nous verrons plus loin des utilisations plus avancées des JSP que l'on considère souvent comme de meilleures pratiques.

`HttpSession session`

La session de l'utilisateur.

`ServletContext application`

L'application web.

`javax.servlet.jsp.PageContext pageContext`

Un objet dont l'utilisation première est de rendre abstraite l'implémentation du serveur, mais servant parfois à partager directement des variables entre des pages JSP et supportant les beans et les balises.

Vous noterez que les classes fondamentales des JSP résident dans le paquetage `javax.servlet.jsp`.

L'*exemple 18-1* montre une page JSP simple qui affiche « Bonjour » à l'aide des variables prédéfinies `request` et `out`. Si votre serveur supporte les JavaServer Pages et si vous souhaitez tester cette page, vous devez placer ce fichier sous la racine des documents du serveur et lui donner l'extension *.jsp*. En supposant que vous ayez sauvé la page sous le nom *hello1.jsp*, vous pouvez y accéder par l'URL *http://serveur:port/hello1.jsp* ou si vous avez placé le fichier dans le chemin de contexte *jsp*, l'URL est alors *http://serveur:port/jsp/hello1.jsp*.

Exemple 18-1. Afficher « Bonjour » avec les JSP

```
<HTML>
<HEAD><TITLE>Bonjour</TITLE></HEAD>
<BODY>
<H1>
<%
if (request.getParameter("name") == null) {
   out.println("Bonjour tout le monde");
}
else {
  out.println("Bonjour, " + request.getParameter("name"));
}
%>
</H1>
</BODY></HTML>
```

Une copie d'écran est montrée à la *figure 18-1*.

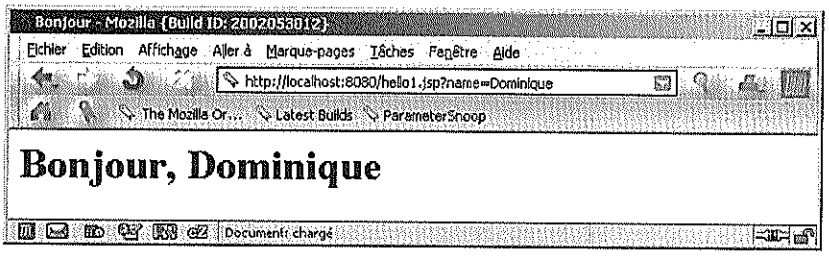


Figure 18-1. Souhaiter le bonjour avec les JavaServer Pages

En coulisses

Comment les JSP fonctionnent-t-elles ? Le serveur crée, compile, charge et exécute automatiquement une servlet spéciale pour générer le contenu de la page, comme le montre la *figure 18-2*. Vous pouvez voir cette servlet spéciale comme une servlet de fond fonctionnant en arrière-plan. Les portions statiques de la page HTML sont générées par la servlet de fond en utilisant des appels à `out.println()`, tandis que les portions dynamiques sont incluses directement. Par exemple, la servlet montrée à l'*exemple 18-2* pourrait être la servlet d'arrière-plan pour *hello1.jsp* s'exécutant sous Tomcat.[2]

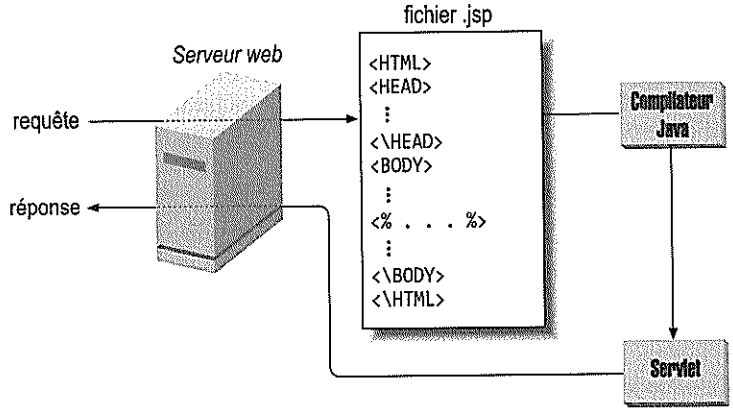


Figure 18-2. Générer les JavaServer Pages

Exemple 18-2. La servlet générée pour hello1.jsp

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import org.apache.jasper.runtime.*;
import org.apache.jasper.*;

public class _0002fhello_00031_0002ejsphello1_jsp_0 extends HttpJspBase {
    private static boolean _jspx_inited = false;

    public _0002fhello_00031_0002ejsphello1_jsp_0() { }
```

2. Si vous souhaitez examiner le vrai code source de la servlet correspondant à une page JSP, regardez dans le répertoire temporaire spécifié dans l'attribut de contexte `javax.servlet.context.tempdir` (voir le chapitre 4, *Retrouver l'information*). Si vous le trouvez, vous avez de grande chance de voir qu'il est bien plus complexe que ce que l'on montre ici.

Exemple 18-2. La servlet générée pour hello1.jsp (suite)

```
    public final void _jspx_init() throws JasperException { }

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
                                throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {
            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                        "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\hello1.jsp";from=(0,0);to=(4,0)]
            out.write("<HTML>\r\n<HEAD><TITLE>Bonjour</TITLE></HEAD>\r\n
                      <BODY>");
            out.write("\r\n<H1>\r\n");
            // end

            // begin [file="C:\\hello1.jsp";from=(4,2);to=(11,0)]
            if (request.getParameter("name") == null) {
               out.println("Bonjour tout le monde");
            }
            else {
              out.println("Bonjour, " + request.getParameter("name"));
            }
            // end

            // HTML // begin [file="C:\\hello1.jsp";from=(11,2);to=(15,0)]
            out.write("\r\n</H1>\r\n</BODY></HTML>\r\n\r\n");
            // end
        } catch (Exception ex) {
```

Exemple 18-2. La servlet générée pour hello1.jsp (suite)

```
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

La première fois que vous accédez à une page JSP, vous noterez sans doute qu'il y a un petit temps de réponse. C'est le temps nécessaire au serveur pour créer et compiler la servlet d'arrière-plan, ce que l'on appelle la *pénalité de la première personne*. Les requêtes suivantes devraient être plus rapides car le serveur peut alors réutiliser la servlet en mémoire. La seule exception est lorsque le fichier *.jsp* change, auquel cas le serveur le remarque et recompile la nouvelle servlet d'arrière-plan. S'il y a une erreur à la compilation, vous pouvez vous attendre à un rapport sur le problème de la part du serveur, habituellement dans la page retournée au client et/ou un journal du serveur.

Vous pouvez éviter la pénalité de la première personne en précompilant les pages JSP. Cela se fait manuellement en effectuant une requête pour chaque page JSP et en passant la chaîne d'interrogation *?jsp_precompile="true"*. Cette requête demande au serveur de compiler la page, mais sans passer la requête au système de gestion des pages JSP. Les fournisseurs de conteneurs de servlets peuvent rendre ce processus plus automatique et plus aisé.

Expressions et déclarations

En plus des scriptlets, les JavaServer Pages offrent la possibilité de placer du code dans des pages à l'aide des expressions et des déclarations. Une expression JSP commence par `<%=` et se termine par `%>`. Toute expression Java comprise entre ces deux balises est évaluée, le résultat est converti en un `String` et le texte est inclus directement dans la page. Cette technique élimine les appels à `out.println()`. Par exemple, `<%= foo %>` inclut la valeur de la variable `foo`.

Une déclaration commence par `<%!` et se termine par `%>`. Entre ces balises, vous pouvez inclure tout code de servlet qui doit être placé en dehors de la méthode de service de la servlet. Vous pouvez déclarer des variables statiques ou d'instance, ou définir de nouvelles méthodes. L'*exemple 18-3* montre une page JSP qui se sert d'une déclaration définissant la méthode `getName()` et d'une expression pour l'affichage. Le commentaire qui se trouve au début de la page montre l'utilisation des balises de commentaire `<%--` et `--%>`.

Exemple 18-3. Souhaiter le bonjour avec une déclaration JSP

```
<%-- hello2.jsp --%>
<HTML>
<HEAD><TITLE>Bonjour</TITLE></HEAD>
<BODY>
<H1>
```

Exemple 18-3. Souhaiter le bonjour avec une déclaration JSP (suite)

```
Bonjour, <%= getName(request) %>
</H1>
</BODY>
</HTML>

<%!
private static final String DEFAULT_NAME = "tout le monde";

private String getName(HttpServletRequest req) {
  String name = req.getParameter("name");
  if (name == null)
    return DEFAULT_NAME;
  else
    return name;
}
%>
```

Cette page JSP se comporte de la même manière que *hello1.jsp*.

Les méthodes spéciales `jspInit()` et `jspDestroy()` peuvent être implémentées au sein d'une déclaration. Elles sont appelée par les méthodes `init()` et `destroy()` de la servlet d'arrière-plan donnant à la page JSP l'opportunité de déclarer du code qui doit être exécuté au moment de l'initialisation et de la destruction. Une page JSP ne peut surcharger les méthodes `init()` et `destroy()` standard car elles sont déclarées finales par la servlet d'arrière-plan.

Directives

Une directive JSP permet à la page JSP de contrôler certains aspects de sa servlet correspondante. Les directive peuvent être utilisées de manière à ce que la servlet définisse son type de contenu, qu'elle importe un paquetage, qu'elle contrôle la mise en tampon de sa sortie et la gestion des sessions, qu'elle étende une classe mère différente ou qu'elle implémente une gestion d'erreurs spéciale. Une directive peut même spécifier l'utilisation d'un langage de script autre que Java.

La syntaxe des directives impose un nom de directive et une paire de nom/valeur, le tout entouré des balises `<%@ %>`. Les guillemets autour de la valeur de l'attribut sont obligatoires :

```
<%@ nomDirective nomAttribut="valeurAttribut" %>
```

La directive page permet à la page JSP de contrôler la servlet générée au travers d'attributs de configuration spéciaux. En voici la liste :

`contentType`

Spécifie le type du contenu de la page générée. Par exemple :

```
<%@ page contentType="text/plain" %>
```

Par défaut le type du contenu est `text/html; charset=8859_1`.

`import`

Spécifie une liste de classes et de paquetages que la servlet doit importer. Il est possible de préciser une liste de plusieurs classes séparées par des virgules. Par exemple :

```
<%@ page import="java.io.*,java.util.Hashtable" %>
```

La liste d'inclusion implicite est `java.lang.*,javax.servlet.*,javax.servlet.http.*,javax.servlet.jsp.*`.

`buffer`

Spécifie, en kilo-octets, la taille minimale obligatoire du tampon de réponse, similaire à la méthode de servlet `setBufferSize()`. La valeur doit être donnée sous la forme ##kb. La valeur spéciale none indique que le contenu doit passer directement au `PrintWriter` de `ServletResponse` (qui peut ou non passer directement le contenu au client). Par exemple :

```
<%@ page buffer="12kb" %>
```

La taille par défaut est 8kb.

`autoFlush`

Spécifie si le tampon doit être vidé lorsqu'il est rempli ou si une exception `IOException` doit être lancée à ce moment-là. La valeur `true` indique qu'il est vidé, `false` indique qu'une exception est lancée. Par exemple :

```
<%@ page autoFlush="true" %>
```

La valeur par défaut est `true`.

`session`

Indique que la page veut avoir accès à la session de l'utilisateur. La valeur `true` place la variable `session` dans la portée et peut définir un cookie client pour gérer la session. La valeur `false` désactive l'accès à la variable `session`. Par exemple :

```
<%@ page session="false" %>
```

La valeur par défaut est `true`.

`errorPage`

Spécifie une page qui sera affichée lorsque qu'un `Throwable` est lancé à partir de la page JSP et s'il n'est pas intercepté avant d'arriver au serveur. Cela se révèle utile car il est difficile de placer des blocs `try/catch` au sein des pages JSP. Par exemple :

```
<%@ page errorPage="/error.jsp" %>
```

Le comportement par défaut dépend de l'implémentation. Le chemin étant relatif au contexte, vous n'avez pas besoin de vous préoccuper de l'ajout du chemin de contexte courant. La cible peut être une page JSP, mais ce n'est pas obligatoire. Si la cible est une servlet, elle peut retrouver le `Throwable` à partir de l'attribut de contexte `javax.servlet.jsp.jspException`.

`isErrorPage`

Indique si la page est destinée à devenir la cible d'un `errorPage`. Si la valeur est `true`, la page peut accéder à la variable implicite `exception` pour retrouver le `Throwable`.

`language`

Spécifie le langage de script utilisé dans les parties de code de la page. Le langage utilisé doit suffisamment interagir avec Java pour que les objets Java nécessaires puissent être vus dans l'environnement de script. Par exemple :

```
<%@ page language="javascript" %>
```

La valeur par défaut est `java`, le seul langage reconnu par la spécification.

Utiliser les directives

L'*exemple 18-4* montre une page JSP nommée *errorMaker.jsp* qui utilise plusieurs directives. Tout d'abord, elle donne la valeur `false` à l'attribut `session` car la page ne se sert pas de l'objet de session et il n'y a aucune raison pour que le serveur créer une session inutile. Ensuite, elle met l'attribut `errorPage` à la valeur `/errorTaker.jsp` afin que, si la page lance une exception non interceptée, la page *errorTaker.jsp* prenne en charge l'affichage du message d'erreur. Le corps de la page est simple. Il lance une exception afin de déclencher le comportement de `errorPage`. (Nous verrons plus loin pourquoi la vérification `if` est employée.)

Exemple 18-4. Fondamentalement mauvais

```
<%-- errorMaker.jsp --%>

<%@ page session="false" %>    <%-- Ne pas envoyer des cookies inutiles. --%>
<%@ page errorPage="/errorTaker.jsp" %>   <%-- Page de gestion d'erreur. --%>

<%-- Nous ne savons que lancer une exception. --%>
<%
  if (System.currentTimeMillis() > 0) {
    throw new Exception("oups");
  }
%>
```

La page *errorTaker.jsp* est montrée à l'*exemple 18-5*. Elle met l'attribut `isErrorPage` à `true` pour signaler qu'elle gère les erreurs et elle emploie l'attribut `import` pour importer les classe de `com.oreilly.servlet`. La page JSP utilise une scriptlet pour définir le code d'état de la réponse à `500`, puis elle écrit dans le corps de la page le nom de classe d'exception, une trace de la pile et un message suggérant de contacter le webmestre afin de signaler le problème.

Exemple 18-5. Votre problème est...

```
<%-- errorTaker.jsp --%>

<%@ page isErrorPage="true" %>
<%@ page import="com.oreilly.servlet.*" %>

<% response.setStatus(500); %>

<HTML>
<HEAD><TITLE>Erreur : <%= exception.getClass().getName() %></TITLE></HEAD>
```

Exemple 18-5. Votre problème est... (suite)

```
<BODY>

<H1>
<%= exception.getClass().getName() %>
</H1>

Nous avons rencontré une erreur pendant l'exécution de votre page :

<PRE>
<%= ServletUtils.getStackTraceAsString(exception) %>
</PRE>

<% String name = request.getServerName(); %>
Veuillez contacter le <A HREF="mailto:webmaster@<%= name %>">
webmestre@<%= name %></A> pour signaler le problème.

</BODY>
</HTML>
```

À chaque requête de *errorMaker.jsp*, une exception est lancée, le contrôle est passé à *errorTaker.jsp* et *errorTaker.jsp* affiche une jolie page d'erreur. Voir la *figure 18-3*.

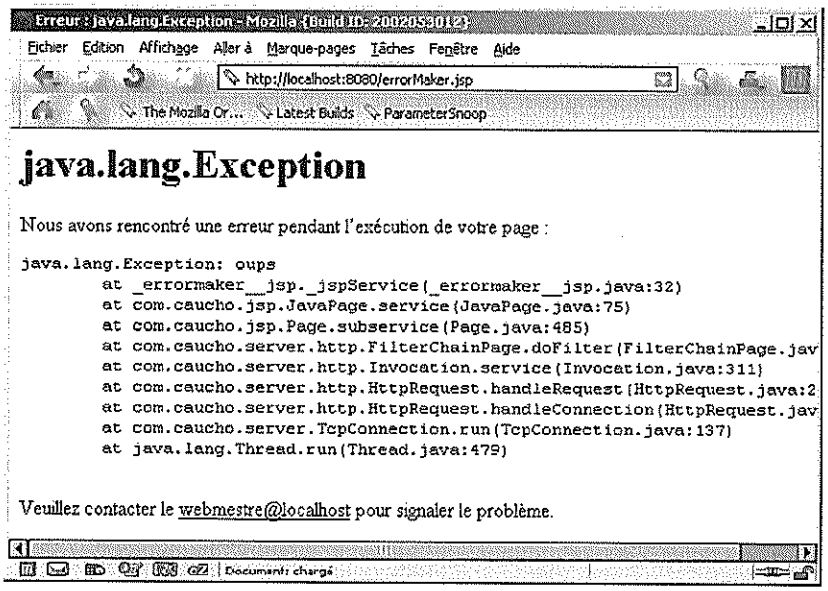


Figure 18-3. Une page d'erreur personnalisable

Pourquoi avons-nous placé un appel à `System.currentTimeMillis()` dans *errorMaker.jsp* ? Cela vient du fait que les pages JSP doivent préserver tous les espaces du texte du corps du document. Cela permet aux JSP d'être utilisées non seulement avec du HTML mais aussi avec du XML où chaque espace peut être très important. Malheu-

reusement, la préservation des espaces signifie que la simple scriptlet :

```
<% throw new Exception("oups"); %>
```

doit générer du code pour afficher une nouvelle ligne après le lancement, du code qui n'est jamais atteint ! La compilation de cette scriptlet sur Tomcat génère le message d'erreur suivant :

```
org.apache.jasper.JasperException: Unable to compile class for
  JSPC:\engines\jakarta-tomcat\work\localhost_8080%2Fjsp\
  _0002ferrorMaker_0002ejsperrorMaker_jsp_3.java:75:
Statement not reached.
                out.write("\r\n");
                ^
```

En contrôlant si `System.currentTimeMillis()` est supérieur à zéro, le compilateur suppose qu'il est possible que l'exception ne soit pas lancée et qu'il n'y a donc pas d'erreur de compilation. Des problèmes similaires se produisent avec des instructions `return` et `throw` au sein des scriptlets.

En fait, il existe plusieurs points de ce type lorsque l'on utilise des scriptlets dans les JSP. Si vous écrivez accidentellement une scriptlet à la place d'une expression (en oubliant le signe égal), déclarez une variable statique dans une scriptlet (où c'est interdit), oubliez un point-virgule (ils ne sont pas obligatoires dans les expressions, mais le sont dans les scriptlets), ou écrivez autre chose que du code Java parfait, vous aurez toutes les chances d'obtenir un message d'erreur confus car le compilateur travaille sur du code Java généré, non sur le fichier JSP. Pour voir ce problème, remplacez `<% name %>` par `<%= name %>` dans *errorTaker.jsp*. Tomcat génère alors l'erreur suivante :

```
org.apache.jasper.JasperException: Unable to compile class for
  JSPC:\engines\jakarta-tomcat\work\localhost_8080%2Fjsp\
  _0002ferrorTaker_0002ejsperrorTaker_jsp_6.java:91:
Class name not found.
                name
                ^
```

Le débogage d'une telle erreur demande souvent au programmeur d'examiner le code généré afin de déterminer ce qui a provoqué l'erreur.

Éviter le code Java dans les pages JSP

Les scriptlets, les expressions et les déclarations permettent de placer du code Java dans une page JSP. Ce sont des outils puissants, encore que nous en décourageons l'emploi. Même dans les mains d'un programmeur avancé, ce type de développement reste une programmation « d'occasion ». Vous écrivez du code Java mais indirectement. Lorsque des erreurs surviennent, ce niveau d'indirection supplémentaire augmente la difficulté de diagnostic du problème.

Entre les mains de non-programmeurs, la situation devient encore pire. Ces personnes doivent apprendre Java, un langage qui n'est pas conçu pour écrire des scripts et qui ne leur est pas destiné. Un langage comme JavaScript, plus simple, plus tolérant et mieux connu dans concepteurs de sites web, est préférable à Java, mais le support de JavaScript n'est pas standard et n'est pas très largement implémentée.

Pour finir, c'est une question de conception. Des concepteurs et des développeurs travaillant sur le même fichier crée un goulot d'étranglement et de nouvelles opportunités d'erreurs. Le contenu et la présentation doivent être séparés. Pour cela et pour déplacer le code Java en dehors de la page, les JSP autorisent l'emploi des JavaBeans et des bibliothèques de balises personnalisées. Nous commençons par les JavaBeans.

JSP et JavaBeans

Une manière très intéressante et très puissante d'utiliser les JavaServer Pages consiste à les faire coopérer avec les composants JavaBeans. Les JavaBeans sont des classes Java réutilisables dont les méthodes et les variables suivent des conventions de nommage spécifiques afin de leur ajouter certaines possibilités. Ils peuvent être inclus directement dans une page JSP en utilisant la balise `<jsp:useBean>`, ou par ce que les JSP appellent une *action*. Un composant JavaBean peut effectuer une tâche bien définie (exécuter des requêtes sur une base de données, se connecter un serveur de courrier, maintenir des informations sur le client, etc.) et rendre l'information résultante disponible à la page JSP au travers de méthodes d'accès simples. Pour plus d'informations sur les JavaBeans, consultez le site *http://java.sun.com/bean* et le livre *Developing Java Beans* de Robert Englander (O'Reilly & Associates).

La différence entre un composant JavaBeans embarqué dans une page JSP et une classe normale utilisée par la servlet générée est que le serveur web peut apporter un traitement spécial au JavaBeans. Par exemple, un serveur peut définir automatiquement des propriétés d'un bean (principalement des variables d'instance) en utilisant les valeurs des paramètres de la requête du client. Autrement dit, si la requête a un paramètre `name` et que le serveur détecte *via* l'introspection (une technique permettant de déterminer, par programmation et à l'exécution, les méthodes et les variables d'une classe Java) que le bean a une propriété `name` et une méthode `setName(String name)`, le serveur peut appeler automatiquement `setName()` avec la valeur du paramètre `name`. Il n'est pas nécessaire d'utiliser la méthode `getParameter()`.

Le serveur peut aussi gérer automatiquement la portée du bean. On peut affecter un bean à une page spécifique (où il est utilisé une fois puis détruit), à une requête spécifique (similaire à la portée de page excepté que le bean persiste entre les appels internes d'inclusion et de redirection), à une session d'un client (il est devient automatiquement disponible chaque fois que le même client se connecte) ou à une application (où le même bean est disponible à l'application web complète).

Embarquer un bean

Les beans sont embarqués dans une page JSP en utilisant l'action `<jsp:useBean>`. Voici sa syntaxe (sensible à la casse et guillemets obligatoires) :

```
<jsp:useBean id="nom" scope="page|request|session|application"
             class="nomClasse" type="nomType">
</jsp:useBean>
```

Vous pouvez définir les attributs suivants dans l'action `<jsp:useBean>` :

`id`

Spécifie le nom du bean. Il s'agit de la clé sous laquelle le bean est sauvegardé si sa portée dépasse celle de la page. S'il existe déjà une instance de bean sauvée sous ce nom, cette instance est utilisée avec cette page. Sinon, un nouveau bean est créé. Par exemple :

```
id="userPreferences"
```

`scope`

Spécifie la portée de la visibilité du bean. La valeur doit être page, `request`, `session` ou `application`. Pour `page`, la variable est créée comme une variable d'instance ; pour `request`, la variable est stockée dans un attribut de requête ; pour `session`, le bean est stocké dans la session de l'utilisateur ; pour `application`, le bean est stocké dans le contexte de servlet. Par exemple :

```
scope="session"
```

La valeur par défaut est page.

`class`

Spécifie le nom de classe du bean. Il sert au moment de la construction du bean. Le nom de classe doit être complet. Par exemple :

```
class="com.company.tracking.UserPreferencesImpl"
```

L'attribut `class` n'est pas obligatoire si le bean existe déjà dans la portée courante, mais l'attribut type doit quand même être donné afin que le système puisse transtyper l'objet dans le bon type. Si un problème survient lors de la construction de la classe indiquée, la page JSP lance une `InstantiationException`.

`type`

Spécifie le type du bean tel qu'il doit être détenu par le système. Il est utilisé pour le transtypage de l'objet quand celui-ci est pris dans la requête, la session ou le contexte. La valeur du type doit être le nom complet d'une classe mère ou d'une interface de la classe réelle. Par exemple :

```
type="com.company.tracking.UserPreferences"
```

S'il n'est pas donné, la valeur par défaut est celle de l'attribut `class`. Si le type ne orrespond pas au type réel de l'objet, la page JSP lance une `ClassCastException`.

`beanName`

L'attribut `class` peut être remplacé par un attribut `beanName`. La différence est que `beanName` utilise `Beans.instantiate()` pour créer l'instance du bean, qui recherche une version sérialisée du bean (un fichier *.ser*) avant de créer une instance initiale. Cela permet d'utiliser des beans préconfigurés. Consultez la documentation Javadoc de `Beans.instantiate()` pour plus d'informations. Par exemple :

```
beanName="com.company.tracking.UserPreferencesImpl"
```

Le corps de l'élément `<jsp:useBean>`, c'est-à-dire tout ce qui se trouve entre les balises `<jsp:useBean>` et `</jsp:useBean>`, est interprété après la création du bean. Si le bean n'est pas créé (car une instance a été trouvée dans la portée indiquée), le corps est ignoré. Par exemple :

```
<jsp:useBean id="prefs" class="com.company.tracking.UserPreferencesImpl">
Si vous voyez ceci, c'est qu'un nouveau bean a été créé !
   </jsp:useBean>
```

Si le corps n'est pas nécessaire, le raccourci XML pour une valise vide, />, peut être employé :

```
<jsp:useBean id="prefs" class="com.company.tracking.UserPreferencesImpl" />
```

Contrôler les propriétés d'un bean

L'action <jsp:setProperty> permet de fixer automatiquement (*via* l'introspection) les propriétés des beans embarqués dans une page à l'aide des valeurs des paramètres de requête. Cela apporte aux beans la possibilité d'accéder aux paramètres de la requête sans avoir à appeler getParameter(). Il y a plusieurs manières d'utiliser cette fonctionnalité. Tout d'abord :

```
<jsp:setProperty name="nomBean" property="*" />
```

Cette ligne indique que tout paramètre de requête dont le nom et le type correspond à une propriété du bean doit être utilisé pour fixer la valeur de cette propriété. Par exemple, si un bean offre une méthode setSize(int size) et que la requête a un paramètre size ayant pour valeur 12, le serveur appelle automatiquement bean.setSize(12) au début du traitement de la requête. Si la valeur du paramètre ne peut être convertie dans le bon type, le paramètre est ignoré. (Notez que si la valeur du paramètre est une chaîne vide, il est traité comme un paramètre inexistant et une chaîne vide n'est pas affectée à la propriété String.)

Notez que l'action utilise le raccourci XML pour une balise vide. C'est très important. La spécification des JSP impose que toutes les actions JSP soient du XML bien formé, même lorsqu'elles se trouvent dans des fichiers HTML. Voici la deuxième utilisation de <jsp:setProperty> :

```
<jsp:setProperty name="nomBean" property="nomPropriété" />
```

Celle ligne indique que la propriété indiquée doit être fixée, s'il existe un paramètre de requête de même nom et de même type. Par exemple, pour fixer uniquement la propriété size, utilisez l'action suivante :

```
<jsp:setProperty name="prefs" property="size" />
```

Voici une troisième utilisation de <jsp:setProperty> :

```
<jsp:setProperty name="nomBean" property="nomPropriété"
param="nomParamètre" />
```

Cette ligne indique que la propriété donnée doit être fixée, s'il existe un paramètre de requête de nom indiqué et de même type. Cela permet d'utiliser un paramètre dont le nom est différent de celui de la propriété. Par exemple, pour fixer la propriété size avec le paramètre fontSize :

```
<jsp:setProperty name="prefs" property="size" param="fontSize" />
```

Voici la dernière utilisation :

```
<jsp:setProperty name="nomBean" property="nomPropriété"
value="constante" />
```

Cette ligne indique que la propriété donnée doit être fixée à la valeur donnée, si nécessaire convertie dans le bon type. Par exemple, pour fixer la taille par défaut à 18 avec une surcharge de paramètre :

```
<jsp:setProperty name="prefs" property="size" value="18" />
<jsp:setProperty name="prefs" property="size" param="fontSize" />
```

Pour les utilisateurs avancés, l'attribut de valeur n'est pas obligatoirement une constante ; il peut être donné par une expression appelée une *expression d'attribut au moment de la requête*. Par exemple, pour s'assurer que la taille n'est jamais inférieure à 6 :

```
<jsp:setProperty name="prefs" property="size" param="fontSize" />
<jsp:setProperty name="prefs" property="size"
    value="<%= Math.max(6, prefs.getSize()) %>" />
```

Enfin, l'action `<jsp:getProperty>` fournit un mécanisme permettant de retrouver les valeurs des propriétés sans utiliser de code Java dans la page. Son utilisation ressemble beaucoup à celui de `<jsp:setProperty>` :

```
<jsp:getProperty name="nomBean" property="nomPropriété" />
```

Cette ligne indique d'inclure, à l'endroit où elle se trouve, la valeur de la propriété donnée du bean indiqué. C'est plus long que d'entrer une expression, mais cela permet d'éviter de placer du code Java dans la page. Le choix entre `<jsp:getProperty>` et une expression est une question de goût personnel. Notez que `<jsp:getProperty>` comporte la valeur de la propriété. Si elle contient des caractères spéciaux pour HTML, il y aura des problèmes. Le projet Apache Struts dispose d'une solution basée sur une balise `<property>` personnalisée, comme nous le verrons plus loin.

Souhaiter le bonjour à l'aide d'un bean

L'*exemple 18-6* montre l'emploi d'un composant JavaBeans avec une page JSP ; il affiche « Bonjour » à l'aide d'un `HelloBean`.

Exemple 18-6. Souhaiter le bonjour à l'aide d'un JavaBean

```
<%-- hello3.jsp --%>

<%@ page import="HelloBean" %>

<jsp:useBean id="hello" class="HelloBean">
  <jsp:setProperty name="hello" property="*" />
</jsp:useBean>

<HTML>
<HEAD><TITLE>Bonjour</TITLE></HEAD>
<BODY>
<H1>
Bonjour, <jsp:getProperty name="hello" property="name" />
</H1>
</BODY>
</HTML>
```

Comme vous pouvez le voir, l'emploi d'un composant JavaBeans dans les JavaServer Pages peut réduire la quantité de code placé dans la page. La classe `HelloBean` contient la logique métier permettant de déterminer le nom de l'utilisateur, la page JSP agissant comme un modèle.

Le code de HelloBean est montré à l'*exemple 18-7*. Son fichier de classe doit être placé dans le répertoire standard des classes utilitaires (*WEB-INF/classes*).

Exemple 18-7. La classe HelloBean

```
public class HelloBean {
  private String name = "tout le monde";

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

C'est peut-être le bean le plus simple que vous pourrez voir. Il dispose d'une seule propriété, name, dont la valeur est fixée à l'aide de setName() et retrouvée à l'aide de getName(). La valeur par défaut de name est `tout le monde`, mais quand la requête inclut un paramètre name, la propriété est automatiquement fixée par le serveur par un appel setName(). Pour tester ce mécanisme, visitez l'URL *http://serveur:port/hello3.jsp*. Vous devriez obtenir quelque chose de similaire à la copie d'écran montrée à la *figure 18-4*.

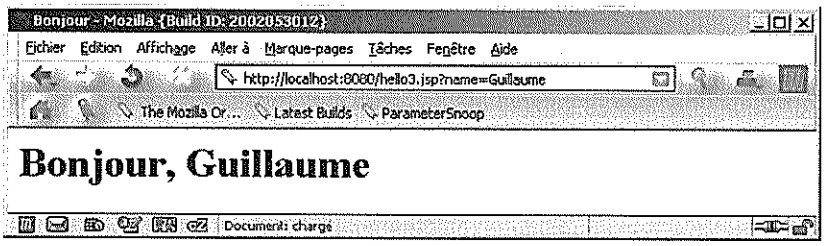


Figure 18-4. Souhaiter le bonjour en utilisant les JavaServer Pages et un composant JavaBeans

Attention à une chose : sur certains serveurs (dont Tomcat 3.2), si un bean se trouve dans la portée `session` ou `application` et si vous modifiez l'implémentation de la classe du bean, vous pourriez recevoir une `ClassCastException` lors d'une prochaine requête. Cette exception survient car le code de la requête généré doit transtyper l'instance du bean quand elle est prise dans la session ou l'application et l'ancien type de bean stocké dans la session ou l'application ne correspond plus au nouveau type de bean. La solution la plus simple consiste à redémarrer le serveur.

Inclusions et renvois

Par le biais d'une combinaison de directives et d'actions, les JSP supportent les inclusions et les renvois. Il existe deux sortes d'inclusions et une sorte de renvoi. La première inclusion se fait avec la directive `include` :

```
<%@ include file="cheminDuFichier" %>
```

Cette inclusion se fait au *moment de la conversion*. Cela signifie qu'elle se produit pendant la création de la servlet d'arrière-plan. Tout le contenu du fichier externe est inclus comme s'il était entré directement dans la page JSP. Pour les programmeurs C, cela fonctionne comme un `#include`. La racine du chemin du fichier est la racine du contexte (*/index.jsp* fait donc référence à *index.jsp* dans l'application web courante) et le chemin ne doit pas sortir de la racine du contexte (n'essayez donc pas *../../../autreContexte*). Le contenu du fichier est en général un fragment de page, non une page complète. Vous pouvez donc employer une extension de fichier telle que *.inc* ou *.jin* pour indiquer ce fait. La page JSP suivante utilise plusieurs directives d'inclusion pour construire une page à partir d'un ensemble de composants (non montrés ici) :

```
<%@ include file = "/header.html" %>
<%@ include file = "/nameCalculation.inc" %>
Bonjour, <%= name %>
<%@ include file = "/footer.html" %>
```

D'où vient la variable `name` ? Elle est créée par le fichier *nameCalculation.inc*. Étant donné que le contenu du fichier est inclus directement, il n'y a pas de séparation de la portée des variables.

La seconde sorte d'inclusion est une action `<jsp:include>` :

```
<jsp:include page="cheminDeLaRessourceDynamique" flush="true" />
```

Cette inclusion se fait au *moment de la requête*. Ce n'est pas une inclusion brute comme la directive `include` ; le serveur exécute la ressource dynamique indiquée et inclut sa sortie au contenu envoyé au client. En coulisses, l'action `<jsp:include>` génère le code qui appelle la méthode `include()` de `RequestDispatcher` et les règles d'utilisation de `include()` s'appliquent aussi à `<jsp:include>` : la page incluse doit être dynamique, elle ne peut définir ni le code d'état ni la valeur des en-têtes et la racine du chemin de la page est celle du contexte.[3] L'attribut `flush` est obligatoire et doit toujours valoir `true` dans les JSP 1.1. Cela indique que le tampon de réponse doit être consommé avant l'inclusion. Étant données les possibilités de l'API Servlet 2.2, la valeur `false` ne peut être acceptée ; les JSP 1.2 construites au-dessus de l'API Servlet 2.3 devraient accepter la valeur `false`.

Par exemple, l'action `<jsp:include>` suivante inclut le contenu généré par un appel à la page *greeting.jsp*. La page *greeting.jsp* pourrait être très similaire à *hello1.jsp*, excepté le HTML englobant qui fait de *hello1.jsp* une page HTML complète :

```
<jsp:include page="/greeting.jsp" />
```

Le corps de la directive `<jsp:include>` peut comporter des balises `<jsp:param>` optionnelles. Ces balises ajoutent des paramètres de requête à la requête d'inclusion. Par exemple, l'action suivante passe un nom par défaut à *greeting.jsp* :

```
<jsp:include page="/greeting.jsp">
  <jsp:param name="defaultName" value="Nouvel utilisateur" />
</jsp:include>
```

3. Il existe une différence mineure entre la méthode `include()` et l'action `<jsp:include>`. La méthode `include()` oblige le chemin de la page à commencer par `/`. L'action `<jsp:include>` accepte aussi les chemins relatifs, comme `../index.jsp`, tant que le chemin ne sort pas du contexte courant.

Pour finir, une page JSP peut renvoyer une requête à l'aide de l'action <jsp:forward> :

```
<jsp:forward page="cheminDeLaRessourceDynamique" />
```

Le renvoi provoque le transfert du contrôle du traitement de la requête à la ressource indiquée. Comme pour l'action <jsp:include>, il est exécuté au moment de la requête et il repose sur la méthode forward() de RequestDispatcher. Les règles d'utilisation de forward() s'appliquent : au moment du renvoi, toute la sortie dans le tampon est effacée et si la sortie a déjà été envoyée, le système lance une exception. L'action suivante renvoie la requête vers une page spéciale si l'utilisateur n'est pas connecté :

```
<% if (session.getAttribute("user") == null) { %>
  <jsp:forward page="/login.jsp" />
<% } %>
```

L'action <jsp:forward> accepte aussi des balises <jsp:param>.

Une application outil

Maintenant que nous avons la possibilité d'exécuter des inclusions et d'embarquer des JavaBeans dans nos pages JSP, examinons comment utiliser ces fonctionnalités pour écrire l'application de visualisation des outils. Nous plaçons un bean dans la page JSP afin de lui donner accès aux informations sur les outils et la page agit comme une vue sur ces données. L'*exemple 18-8* montre la page JSP, appelée *toolview.jsp*.

Exemple 18-8. Une application de visualisation d'outils à l'aide des JSP

```
<%-- toolview.jsp --%>

<%
  String title = "Liste des outils";
  String deck = "Une liste des outils de création de contenu";
  String desc = "Sans les outils, les gens ne sont rien de plus que des " +
                "animaux. Et plutôt faibles, en plus. Voici une liste " +
                "d'outils de création de contenu basé sur les servlets que " +
                "vous pouvez utiliser afin de ne plus être une mauviette.";
%>

<%@ include file="/header.jsp" %>

<%@ page session="false" %>
<%@ page errorPage="/errorTaker.jsp" %>

<jsp:useBean id="toolbean" class="ToolBean" scope="application">
  <jsp:setProperty name="toolbean" property="toolsFile"
                   value='<%= application.getInitParameter("toolsFile") %>' />
</jsp:useBean>

<%
  Tool[] tools = toolbean.getTools(request.getParameter("state"));

  for (int i = 0; i < tools.length; i++) {
```

Exemple 18-8. Une application de visualisation d'outils à l'aide des JSP (suite)

```
    Tool tool = tools[i];
%>
  <HR SIZE=2 ALIGN=LEFT>

  <H3>
  <%= tool.name %>

  <% if (tool.isNewWithin(45)) { %>
    <FONT COLOR=#FF0000><B> (Nouveau !) </B></FONT>
  <% } else if (tool.isUpdatedWithin(45)) { %>
    <FONT COLOR=#FF0000><B> (Mis à jour!) </B></FONT>
  <% } %>

  </H3>
  <A HREF="<%= tool.homeURL %>"><%= tool.homeURL %></A><BR>

  <%= tool.comments %>

<% } %>

<%@ include file="/footer.jsp" %>
```

Nous commençons par définir le titre, le sujet et la description de la page comme des `String` au sein d'une scriptlet. Nous utilisons ensuite la directive `include` pour inclure le contenu de l'en-tête standard à partir de *header.jsp* et le contenu du base de page standard (à la fin du fichier) à partir de *footer.jsp*. La directive `include` place le contenu des fichiers d'en-tête et de base de page dans le fichier courant pendant la phase de traduction. Les variables `title`, `deck` et `desc` sont donc visibles dans les fichiers inclus et, en fait, *header.jsp* utilisent ces variables. Les fichiers *header.jsp* et *footer.jsp* sont montrés à l'*exemple 18-9* et à l'*exemple 18-10*.

Exemple 18-9. Le fichier header.jsp

```
<%-- header.jsp --%>
<%-- Dépend de la présence des variables title, deck et desc --%>

<HTML><HEAD><TITLE><%= title %></TITLE></HEAD>

<BODY BGCOLOR="#FFFFFF" BACKGROUND="/images/background.gif"
      LINK="#003333" ALINK="#669999" VLINK="#333333">

<IMG SRC="/images/banner.gif" WIDTH=600 HEIGHT=87 BORDER=0><BR>

<TABLE>
<TR>
<TD WIDTH=125 ROWS=10 VALIGN=TOP>
 <BR><BR><BR>
 <FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#FF0000">
 <A HREF="/index.html">Accueil</A><BR>
```

Exemple 18-9. Le fichier header.jsp (suite)

```
 <A HREF="/hosting.html">Hébergement</A><BR>
 <A HREF="/engines.html">Moteurs</A><BR>
 </FONT>
</TD>

<TD WIDTH=475>

  <TABLE CELLPADDING=5><TR><TD WIDTH=600 BGCOLOR="#006699" VALIGN=TOP>
  <B><FONT FACE="Arial,Helvetica" SIZE="+2">
  <%= title %>
  </FONT></B>
  </TD></TR></TABLE>

  <B><FONT FACE="Arial,Helvetica" SIZE="+1" COLOR="#003366">
  <%= deck %>
  </FONT></B><P>

  <P>

  <FONT FACE="Arial,Helvetica">

  <%= desc %>
```

Exemple 18-10. Le fichier footer.jsp

```
<%-- footer.jsp --%>
<%-- Ne dépend pas de la présence de certaines variables. --%>

 </FONT>

</TD>
</TR>

<TR>
<TD></TD>
<TD WIDTH=475 ALIGN=CENTER COLSPAN=3>
<HR>
<FONT FACE="Arial,Helvetica">
<A HREF="/index.html">Accueil</A>  
<A HREF="/hosting.html">Hébergement</A>  
<A HREF="/engines.html">Moteurs</A>  <P>
</FONT>

<TABLE WIDTH=100%>
<TR>
<TD WIDTH=260 ALIGN=LEFT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
<A HREF="/copyright.html">Copyright</A> &copy; 2000 Jason Hunter<BR>
Tous droits réservés.</TD>
```

Exemple 18-10. Le fichier footer.jsp (suite)

```
<TD WIDTH=5></FONT></TD>
<TD WIDTH=230 ALIGN=RIGHT VALIGN=TOP>
<FONT FACE="Arial,Helvetica">
Contact : <A HREF="mailto:webmaster@servlets.com">webmaster@servlets.com</a>
</FONT></TD>
</TR>
</TABLE>

</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Ensuite, nous utilisons, dans la page *toolview.jsp*, l'action `<jsp:useBean>` pour embarquer un JavaBean nommé `toolbean` qui est une instance de la classe `ToolBean`. Le bean étant dans la portée de l'application, une seule instance des données sera automatiquement stockée dans le contexte et rendue disponible à toutes les pages.

Dans le corps de la balise `<jsp:useBean>`, nous plaçons une action `<jsp:setProperty>` qui fixe la propriété `toolsFile` du bean à la valeur du paramètre d'initialisation du contexte `toolsFile`. Nous utilisons une expression d'attribut défini au moment de la requête pour extraire la valeur. Le bean sait ainsi à partir de quel fichier sont extraites les informations sur les outils. Il serait préférable que le bean accède directement à la variable de l'application pour retrouver cette information, mais les beans ont uniquement accès à ce que la page JSP fournit. On pourrait passer le nom du fichier dans un paramètre du constructeur du bean, mais les beans sont instanciés en utilisant le constructeur par défaut sans argument ; ce n'est donc pas possible ici. L'approche prise consiste à inclure l'action `<jsp:setProperty>` dans le corps de la balise `<jsp:useBean>` pour qu'à la première construction du bean il reçoive directement le nom du fichier à partir duquel effectuer le chargement.

Il y a deux choses à noter à propos de l'expression d'attribut défini au moment de la requête. Premièrement, nous devons utiliser des guillemets simples autour de la valeur car nous plaçons des guillemets doubles dans l'expression. Cela permet de ne pas perturber l'analyseur syntaxique des JSP. Nous aurions pu aussi appliquer l'échappement aux guillemets doubles dans l'expression en utilisant des barres obliques inverses. Deuxièmement, si le paramètre d'initialisation n'existe pas et que l'appel à `getInitParameter()` retourne `null`, une exception `JspException` est lancée car une valeur nulle n'est pas valide dans une expression d'attribut défini au moment de la requête.[4]

Après la balise `<jsp:useBean>`, nous utilisons une scriptlet pour extraire du bean le tableau des outils. On appelle la méthode `getTools()` avec la valeur du paramètre `state`

4. Techniquement, la spécification des JSP ne fait aucun commentaire sur ce qui pourrait se passer lorsque l'expression d'attribut défini au moment de la requête est nulle. L'implémentation de référence Tomcat est donc devenu l'arbitre et Tomcat 3.2 lance une exception.

dans le cas où l'utilisateur ne veut voir que les outils dans un certain état. Il est tentant de vouloir écrire une action `<jsp:setProperty>` qui permettrait au bean d'accéder directement à la valeur du paramètre `state`, mais puisque le bean se trouve dans la portée de l'application, ce n'est pas possible. Si nous le faisions, nous obtiendrions des problèmes de concurrence lorsque plusieurs requêtes tenteraient d'accéder à la même propriété sur l'instance partagée du bean.

Une fois la tableau des outils récupéré, nous employons un autre code de scriptlet pour démarrer une boucle `for` d'itération sur le tableau. Dans la boucle `for`, nous plaçons la logique qui affiche chaque outil individuel. Nous utilisons des expressions pour afficher les valeurs simples comme le nom de l'outil, l'URL et les commentaires. Nous nous servons de scriptlets pour gérer l'affichage des remarques Nouveau et Mis à jour.[5] Le code de la classe `ToolBean` est donné à l'*exemple 18-11*. Le code de la classe `Tool` est le même que celui montré au chapitre 14, *L'environnement Tea*.

Exemple 18-11. La classe utilitaire ToolBean

```
import java.util.*;

public class ToolBean {

  private Tool[] tools;
  private String state;

  public ToolBean() { }

  public void setToolsFile(String toolsFile) throws Exception {
    // Aucun moyen d'accéder directement au contexte de l'application
    // à partir d'un bean.
    tools = Tool.loadTools(toolsFile);
  }

  public Tool[] getTools(String state) throws Exception {
    if (tools == null) {
      throw new IllegalStateException(
        "Vous devez toujours définir la propriété toolsFile d'un ToolBean");
    }

    if (state == null) {
```

5. Du fait des règles de conservation des espaces dans les JSP, nous devons faire attention lorsque l'on écrit les instructions `if/else` dans des scriptlets. Le code suivant ne fonctionne *pas* :

```
<% if (tool.isNewWithin(45)) { %>
  <FONT COLOR=#FF0000><B> (Nouveau !) </B></FONT>
<% } %>
<% else if (tool.isUpdatedWithin(45)) { %>
  <FONT COLOR=#FF0000><B> (Mis à jour !) </B></FONT>
<% } %>
```

Avec ce code, la servlet d'arrière-plan essaie d'afficher un passage à la ligne entre les clauses `if` et `else`, provoquant l'erreur de compilation étrange `'else' without 'if'`.

Exemple 18-11. La classe utilitaire ToolBean (suite)

```
      return tools;
    }
    else {
      // Ne retourner que les outils correspondant au "state" donné.
      List list = new LinkedList();
      for (int i = 0; i < tools.length; i++) {
        if (tools[i].getStateFlag().equalsIgnoreCase(state)) {
          list.add(tools[i]);
        }
      }
      return (Tool[]) list.toArray(new Tool[0]);
    }
  }
}
```

La méthode `setToolsFile()` est appelée automatiquement par l'action `<jsp:setProperty>` juste après la construction du bean. Elle charge les informations sur les outils à partir du fichier indiqué.

La méthode `getTools()` est appelée par la page JSP directement au sein d'une scriptlet. La méthode prend en paramètre l'état de l'outil avec lequel se fait la comparaison ; si l'état est `null`, elle retourne la liste complète. Si la variable `tools` vaut `null`, `setToolsFile()` n'a pas dû se terminer correctement et le bean lance une `IllegalStateException` pour signaler le problème à l'utilisateur.

Bibliothèques de balises personnalisées

Pour conclure, examinons un des aspects les plus intéressants des JavaServer Pages : le support des *bibliothèques de balises personnalisées*. Les balises personnalisées (ou les *actions personnalisées*) permettent à une page JSP de contenir des balises XML qui, bien qu'elles ressemblent à du HTML, provoquent l'exécution d'un code Java spécifique. Les possibilités d'utilisation des bibliothèques de balises personnalisées sont plutôt attirantes. Par exemple, Live Software (les créateurs de JRun) se sert des balises personnalisées pour implémenter une version 100% portable des balises d'Allaire Cold Fusion appelée CF_Anywhere. (Peu après, Live Software a été acheté par Allaire.)

Les détails de l'écriture d'une bibliothèque de balises personnalisées sont assez complexes et dépassent le propos de ce livre. Heureusement, plusieurs bibliothèques de balises en *open source* sont en cours de développement et prouvent déjà leur utilité. Les deux plus importantes sont Apache Taglibs et Apache Struts, toutes deux du projet Apache Jakarta à *http://jakarta.apache.org*. Apache Taglibs contient des bibliothèques de balises à but général ; Apache Struts contient une bibliothèque de balises destinée au support d'une architecture de type Modèle 2, mais la plupart des balises sont généralement utiles. Pour une liste des bibliothèques de balises JSP, visitez le site *http://jsptags.com*. Il y a aussi en développement une bibliothèque de balises standard sous forme d'un processus de demande de spécification Java formelle, le JSR-052. Pour plus d'informations sur JSR-052, consultez *http://java.sun.com/aboutJava/communityprocess/jsr/jsr_052_jsptaglib.html*.

Utiliser les bibliothèques de balises personnalisées

Les actions personnalisées permettent d'embarquer une logique applicative dans des pages JSP à l'aide de balises de type HTML. Avec le bon ensemble de balises, la logique qui était précédemment écrite avec des scriptlets est maintenant écrite avec des balises. Par exemple, le projet Apache Struts dispose d'une balise `<iterate>` que l'on utilise pour remplacer une boucle `for`. La balise `<iterate>` exécute le contenu de son corps une fois pour chaque élément d'une collection (`List`, `Set`, `Vector`, `Map`, tableau, etc.). À chaque itération, elle place l'élément dans la portée de la page afin qu'il puisse servir dans le corps.[6] Struts offre aussi une balise `<property>` qui fonctionne comme `<jsp:getProperty>` avec en plus la possibilité de filtrer le contenu en vue de son affichage HTML ; les caractères HTML spéciaux sont convertis en leur entité caractère correspondante. Par exemple, < devient `%lt;`.

L'emploi combiné de `<iterate>` et de `<property>` permet d'écrire l'extrait de page suivant qui affiche tous les cookies envoyés avec la requête, tous les caractères spéciaux étant correctement filtrés :

```
<%@ taglib uri="/WEB-INF/struts.tld" prefix="struts" %>

<struts:iterate id="cookie" collection="<%= request.getCookies() %>">
  <struts:property name="cookie" property="name" /> =
  <struts:property name="cookie" property="value" /> <BR>
</struts:iterate>
```

Pour installer et utiliser une bibliothèque de balises telle que Struts, vous devez effectuer plusieurs opérations :

1. Téléchargez et extrayez la distribution de la bibliothèque de balises. Par exemple, Struts est disponible à *http://jakarta.apache.org*.
2. Placez les classes Java de la bibliothèque de balises de manière à ce que le serveur les trouve. Par exemple, placez le fichier *struts.jar* dans *WEB-INF/lib* ou placez les classes compilées dans *WEB-INF/classes*.
3. Placez le fichier TLD (*Tag Library Descriptor*) quelque part sous le répertoire *WEB-INF*. On utilise souvent, soit *WEB-INF/tlds*, soit *WEB-INF* directement, l'emplacement exact n'ayant pas d'importance. Par exemple, placez *struts.tld* dans *WEB-INF*. Le fichier TLD est un fichier de données XML donnant au serveur les informations sur chaque balise de la bibliothèque : son nom, sa classe, comment utiliser ses attributs, comment utiliser son corps, et ainsi de suite. Les TLD ont aussi un autre intérêt : il est possible de générer une page web documentant la bibliothèque de balises à l'aide du fichier TLD (peut-être en employant une feuille de style XSL).
4. Ajoutez une entrée `<taglib>` au fichier *web.xml*. Puis, dans cette entrée, placez une entrée `<taglib-uri>` donnant le nom de recherche de la bibliothèque de balises et une entrée `<taglib-location>` donnant l'emplacement du TLD. Cette étape ne serait pas nécessaire si l'entrée `<taglib-uri>` correspondait exactement à l'entrée `<taglib-location>`. Par exemple :

6. La balise `<iterate>` a besoin du JDK 1.2 ou ultérieur. Pour le JDK 1.1, il existe une balise `<enumerate>`.

```
<taglib> <!-- Puisque l'uri correspond à l'emplacement, -->
        <!-- ceci est optionnel. -->
    <taglib-uri>/WEB-INF/struts.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts.tld</taglib-location>
</taglib>
```

L'entrée `<taglib-uri>` doit être un chemin d'URI. `struts`, `http://jakarta.apache.org/struts` et `/WEB-INF/struts.tld` sont donc des valeurs valides. En général, un chemin d'URI fait référence à un emplacement de ressources réel, mais il est utilisé ici uniquement comme identifiant de la bibliothèque de balises.

5. Dans chaque page JSP utilisant la bibliothèque de balises, ajoutez la directive `taglib` suivante. Elle indique au serveur de charger la bibliothèque de balises à partir de l'URI de recherche indiqué et de placer ces balises dans le préfixe de l'espace de nommage XML `struts` :

   ```
   <%@ taglib uri="/WEB-INF/struts.tld" prefix="struts" %>
   ```

6. Utilisez les balises dans la page JSP, en vous assurant que le préfixe de l'espace de nommage XML correspond à celui déclaré dans la directive `taglib` :

   ```
   <struts:property name="nomBean" property="nomPropriété" />
   ```

   Il est possible que le créateur d'une bibliothèque de balise fournisse le TLD dans le fichier JAR de la bibliothèque. Dans ce cas, au lieu de pointer sur le *.tld,* il suffit de pointer sur le *.jar* lui-même.

Une application outils utilisant les bibliothèques de balises personnalisées

Avec les balises `<iterate>` et `<property>` de Struts, nous pouvons simplifier notre page *toolview.jsp*. Nous profiterons aussi de cette opportunité pour montrer une architecture Modèle 2 basée sur les servlets dans laquelle une servlet reçoit la requête, ajoute les attributs à l'objet de requête, puis envoie la requête à une page JSP qui joue le rôle de modèle. L'*exemple 18-12* montre la servlet contrôleur.

Exemple 18-12. Un contrôleur de servlet de type Modèle 2

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ToolServlet extends HttpServlet {

  Tool[] tools = null;

  public void init() throws ServletException {
    // Charger les données sur les outils dans l'initialisation pour plus
    // de simplicité.
    String toolsFile =
      getServletContext().getInitParameter("toolsFile"); // De web.xml.
    if (toolsFile == null) {
```

Exemple 18-12. Un contrôleur de servlet de type Modèle 2 (suite)

```
      throw new ServletException(
        "Un fichier de données sur les outils doit être spécifié dans " +
        "le paramètre d'initialisation du contexte toolsFile");
    }
    log("Chargement des outils à partir de " + toolsFile);
    try {
      tools = Tool.loadTools(toolsFile);
      if (tools.length == 0) {
        log("Aucun outil trouvé dans " + toolsFile);
      }
      else {
        log(tools.length + " outil(s) trouvé(s) dans " + toolsFile);
      }
    }
    catch (Exception e) {
      throw new ServletException(e);
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    Tool[] tools = null;

    // Placer un attribut "tools" adéquat dans la requête.
    String state = req.getParameter("state");
    if (state == null) {
      req.setAttribute("tools", getTools());
    }
    else {
      req.setAttribute("tools", getTools(state));
    }

    // Envoyer la requête à la page JSP pour son traitement.
    RequestDispatcher disp = req.getRequestDispatcher("/toolview-tag.jsp");
    disp.forward(req, res);
  }

  public Tool[] getTools() {
    return tools;
  }

  public Tool[] getTools(String state) {
    List list = new LinkedList();
    for (int i = 0; i < tools.length; i++) {
      if (tools[i].getStateFlag().equalsIgnoreCase(state)) {
        list.add(tools[i]);
      }
    }
```

Exemple 18-12. Un contrôleur de servlet de type Modèle 2 (suite)

```
    return (Tool[]) list.toArray(new Tool[0]);
  }
}
```

Cette servlet se comporte de manière similaire à `ToolApp` dans Tea et à `ToolServlet` dans WebMacro. Elle charge les informations sur les outils dans sa méthode `init()` et, à chaque requête, elle place le sous-ensemble adéquat d'outils dans l'objet de requête. La servlet renvoie la requête à une page JSP qui prend en charge l'affichage et cette page JSP accède aux outils dans l'objet de requête à l'aide d'une balise `<jsp:useBean>`. L'*exemple 18-13* montre la page JSP nommée *toolview-tag.jsp*.

Exemple 18-13. L'application outil réécrite avec des bibliothèques de balises

```
<%-- toolview-tag.jsp --%>

<%@ taglib uri="/WEB-INF/struts.tld" prefix="struts" %>

<%
  String title = "Liste des outils";
  String deck = "Une liste des outils de création de contenu";
  String desc = "Sans les outils, les gens ne sont rien de plus que des " +
                "animaux. Et plutôt faibles, en plus. Voici une liste " &
                "d'outils de création de contenu basé sur les servlets que " &
                "vous pouvez utiliser afin de ne plus être une mauviette."
%>

<%@ include file="/header.jsp" %>

<%@ page session="false" %>
<%@ page errorPage="/errorTaker.jsp" %>

<%-- Prendre le tableau des outils dans l'attribut de requête. --%>
<jsp:useBean id="tools" class="Tool[]" scope="request"/>

<struts:iterate id="tool" collection="<%= tools %>">
  <HR SIZE=2 ALIGN=LEFT>

  <H3>
  <%-- Appliquer automatiquement l'échappement HTML aux valeurs. --%>
  <struts:property name="tool" property="name" />

  <% if (((Tool)tool).isNewWithin(45)) { %>
    <FONT COLOR=#FF0000><B> (Nouveau !) </B></FONT>
  <% } else if (((Tool)tool).isUpdatedWithin(45)) { %>
    <FONT COLOR=#FF0000><B> (Mis à jour !) </B></FONT>
  <% } %>

  </H3>
  <A HREF="<struts:property name="tool" property="homeURL"/>">
```

Exemple 18-13. L'application outil réécrite avec des bibliothèques de balises (suite)

```
            <struts:property name="tool" property="homeURL"/></A><BR>

  <%-- Supposer que nous ne voulons pas de HTML dans les commentaires. --%>
  <struts:property name="tool" property="comments" />

</struts:iterate>

<%@ include file="/footer.jsp" %>
```

Cette page commence par utiliser la directive `taglib` pour charger la bibliothèque de balises personnalisées Struts. N'oubliez pas cette étape ! Sinon, vous n'obtiendrez pas directement un message d'erreur, mais la page ne fonctionnera pas. En effet, sans la directive `taglib`, les balises `<struts:iterate>` et `<struts:property>` sont vues par le serveur comme des balises HTML normales et elles sont traitées comme du contenu de page à afficher au lieu d'une logique à exécuter !

Le corps de la page utilise la balise `<struts:iterate>` pour boucler sur le tableau des outils et la balise `<struts:property>` pour afficher le nom, l'URL et les commentaires de chaque outil. Si les commentaires peuvent contenir des balises HTML, il faudrait alors utiliser la balise `<jsp:getProperty>` afin que le HTML ne soit pas filtré.

L'emploi des balises personnalisées de Apache Struts enlève du code de la page HTML, mais il reste quand même quelques scriptlets. Pour supprimer toutes les scriptlets, il faudrait créer une balise personnalisée qui effectue la comparaison d'estampilles. Malheureusement, du fait de tous les détails de l'écriture de balises personnalisées, nous ne pouvons pas inclure un tutoriel ici. Si vous voulez en savoir plus sur la création de balises personnalisées, lisez le livre *JavaServer Pages* de Hans Bergsten (Éditions O'Reilly).

19

Affaires diverses

Chaque maison a son tiroir à fouillis — un tiroir rempli à ras bord d'affaires diverses, qui ne vont dans aucun tiroir organisé, mais ne peuvent être jetées en raison de leur nécessité occasionnelle. Ce chapitre ressemble à ce tiroir. Il contient une foule d'exemples de servlets utiles et des astuces qui ne vont nulle part ailleurs. Vous allez y trouver des servlets qui analysent les paramètres, envoient un e-mail, exécutent des programmes, utilisent des moteurs d'expressions régulières, utilisent des méthodes natives et agissent comme des clients RMI. Il présente aussi les bases des techniques de débogage ainsi que des suggestions pour ajuster les performances des servlets.

Analyser les paramètres

Si vous avez essayé d'écrire à la main vos propres servlets au fur et à mesure de la lecture de ce livre, vous avez probablement noté combien il peut être peu commode d'obtenir et d'analyser les paramètres des requêtes, en particulier lorsque les paramètres doivent être convertis en un format non-`String`. Par exemple, supposons que vous vouliez récupérer le paramètre `count` et sa valeur sous forme d'un `int`. De plus, supposons que vous vouliez gérer les conditions d'erreur en appelant `handleNoCount()` si `count` n'existe pas et `handleMalformedCount()` si `count` ne peut pas être lu comme un entier. Pour cela, l'API Servlet standard nécessite le code suivant :

```
int count;

String param = req.getParameter("count");
if (param == null || param.length() == 0) {
  handleNoCount();
} else {
  try {
    count = Integer.parseInt(param);
  } catch (NumberFormatException e) {
    handleMalformedCount();
  }
}
```

Cela ressemble-t-il à du code que vous avez écrit ? Ce n'est pas très propre, non ? Une meilleure solution est de laisser à une classe utilitaire la responsabilité de récupérer et d'analyser les paramètres. La classe `com.oreilly.servlet.ParameterParser` est justement une telle classe. En utilisant `ParameterParser`, nous pouvons écrire le code précédent plus élégamment :

```
int count;

ParameterParser parser = new ParameterParser(req);
try {
  count = parser.getIntParameter("count");
} catch (NumberFormatException e) {
  handleMalformedCount();
} catch (ParameterNotFoundException e) {
  handleNoCount();
}
```

La méthode `getIntParameter()` de l'analyseur de paramètre retourne la valeur du paramètre spécifié sous forme d'un `int`. Elle lance une `NumberFormatException` si le paramètre ne peut pas être converti en un `int` et une `ParameterNotFoundException` si le paramètre ne fait pas partie de la requête. Elle lance aussi une `ParameterNotFoundException` si la valeur du paramètre est une chaîne vide. Cela arrive souvent lorsque des formulaires soumis comportent des champs dans lesquels rien n'a été entré, ce qui doit être traité comme un paramètre manquant.

Si vous pensez qu'il suffit que la servlet utilise une valeur par défaut lorsqu'il y a un problème avec un paramètre, le code peut même être plus simple :

```
ParameterParser parser = new ParameterParser(req);
int count = parser.getIntParameter("count", 0);
```

Cette seconde version de `getIntParameter()` prend 0 comme valeur par défaut et la retourne au lieu de lancer une exception.

Il est aussi possible de déterminer s'il manque des paramètres obligatoires dans la requête :

```
ParameterParser parser = new ParameterParser(req);
String[] required = { "fname", "lname", "account" };
String[] missing = parser.getMissingParameters(required);
```

La méthode retourne `null` s'il ne manque aucun paramètre.

Et, pour finir, `ParameterParser` supporte l'internationalisation *via* la méthode `setCharacterEncoding()`. Elle indique le jeu de caractère à utiliser pour l'interprétation des valeurs de paramètres. Cette information peut venir d'un cookie utilisateur, d'un champ de formulaire caché ou de la session utilisateur :

```
ParameterParser parser = new ParameterParser(req);
parser.setCharacterEncoding("Shift_JIS");
String japaneseValue = parser.getStringParameter("latinName");
```

En interne, `ParameterParser` utilise l'astuce de `getBytes()` montrée au chapitre 13, *Internationalisation*, pour effectuer la conversion. Les noms de paramètres doivent utiliser le jeu de caractères Latin-1 car la recherche de la valeur se sert des méthodes `getParameter()` et `getParameterValues()` de l'API Servlet qui n'ont pas encore été internationalisées.

Code de ParameterParser

La classe ParameterParser contient une dizaine de méthodes qui retourne les paramètres de la requête, deux pour chaque type primitif de Java. Elle a aussi deux méthodes getStringParameter() au cas où vous voudriez obtenir le paramètre dans son format String. Le code de la classe ParameterParser est donné à l'*exemple 19-1* ; celui de ParameterNotFoundException est à l'*exemple 19-2*.

Exemple 19-1. La classe ParameterParser

```
package com.oreilly.servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;

public class ParameterParser {

  private ServletRequest req;
  private String encoding;

  public ParameterParser(ServletRequest req) {
    this.req = req;
  }

  public void setCharacterEncoding(String encoding)
              throws UnsupportedEncodingException {
    // Tester la validité du codage.
    "".getBytes(encoding);
    // Si l'on arrive ici c'est que le codage est valide, le conserver.
    this.encoding = encoding;
  }

  public String getStringParameter(String name)
      throws ParameterNotFoundException {
    String[] values = req.getParameterValues(name);
    if (values == null) {
      throw new ParameterNotFoundException(name + " non trouvé");
    } else if (values[0].length() == 0) {
      throw new ParameterNotFoundException(name + " est vide");
    } else {
      if (encoding == null) {
        return values[0];
      } else {
        try {
          return new String(values[0].getBytes("8859_1"), encoding);
        } catch (UnsupportedEncodingException e) {
          return values[0];  // Ne doit jamais arriver.
        }
      }
```

Exemple 19-1. La classe ParameterParser (suite)

```
    }
  }

  public String getStringParameter(String name, String def) {
    try { return getStringParameter(name); }
    catch (Exception e) { return def; }
  }

  public boolean getBooleanParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    String value = getStringParameter(name).toLowerCase();
    if ((value.equalsIgnoreCase("true")) ||
        (value.equalsIgnoreCase("on")) ||
        (value.equalsIgnoreCase("yes"))) {
        return true;
    } else if ((value.equalsIgnoreCase("false")) ||
               (value.equalsIgnoreCase("off")) ||
               (value.equalsIgnoreCase("no"))) {
        return false;
    } else {
      throw new NumberFormatException("Paramètre " + name + " valeur " +
                                      value + " n'est pas un booléen");
    }
  }

  public boolean getBooleanParameter(String name, boolean def) {
    try { return getBooleanParameter(name); }
    catch (Exception e) { return def; }
  }

  public byte getByteParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return Byte.parseByte(getStringParameter(name));
  }

  public byte getByteParameter(String name, byte def) {
    try { return getByteParameter(name); }
    catch (Exception e) { return def; }
  }

  public char getCharParameter(String name)
      throws ParameterNotFoundException {
    String param = getStringParameter(name);
    if (param.length() == 0)
      throw new ParameterNotFoundException(name + " est une chaîne vide");
    else
      return (param.charAt(0));
  }

  public char getCharParameter(String name, char def) {
```

Exemple 19-1. La classe ParameterParser (suite)

```
    try { return getCharParameter(name); }
    catch (Exception e) { return def; }
  }

  public double getDoubleParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return new Double(getStringParameter(name)).doubleValue();
  }

  public double getDoubleParameter(String name, double def) {
    try { return getDoubleParameter(name); }
    catch (Exception e) { return def; }
  }

  public float getFloatParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return new Float(getStringParameter(name)).floatValue();
  }

  public float getFloatParameter(String name, float def) {
    try { return getFloatParameter(name); }
    catch (Exception e) { return def; }
  }

  public int getIntParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return Integer.parseInt(getStringParameter(name));
  }

  public int getIntParameter(String name, int def) {
    try { return getIntParameter(name); }
    catch (Exception e) { return def; }
  }

  public long getLongParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return Long.parseLong(getStringParameter(name));
  }

  public long getLongParameter(String name, long def) {
    try { return getLongParameter(name); }
    catch (Exception e) { return def; }
  }

  public short getShortParameter(String name)
      throws ParameterNotFoundException, NumberFormatException {
    return Short.parseShort(getStringParameter(name));
  }
```

Exemple 19-1. La classe ParameterParser (suite)

```
  public short getShortParameter(String name, short def) {
    try { return getShortParameter(name); }
    catch (Exception e) { return def; }
  }

  public String[] getMissingParameters(String[] required) {
    Vector missing = new Vector();
    for (int i = 0; i < required.length; i++) {
      String val = getStringParameter(required[i], null);
      if (val == null) {
        missing.addElement(required[i]);
      }
    }
    if (missing.size() == 0) {
      return null;
    }
    else {
      String[] ret = new String[missing.size()];
      missing.copyInto(ret);
      return ret;
    }
  }
}
```

Exemple 19-2. La classe ParameterNotFoundException

```
package com.oreilly.servlet;

public class ParameterNotFoundException extends Exception {

  public ParameterNotFoundException() {
    super();
  }

  public ParameterNotFoundException(String s) {
    super(s);
  }
}
```

Envoyer un e-mail

Il est parfois nécessaire, ou simplement pratique, qu'une servlet envoie un message e-mail. Par exemple, imaginez une servlet qui reçoit les données à partir d'un formulaire de réponse d'un utilisateur. La servlet pourrait vouloir envoyer les données à une liste de diffusion des personnes intéressées. Ou imaginez une servlet qui rencontre un problème inattendu et qui sait envoyer une page par e-mail à son administrateur afin de demander de l'aide.

Une servlet a quatre possibilités pour envoyer un e-mail :

- Elle prend en charge les détails elle-même : établir une connexion par socket à un serveur de courrier et converser *via* un protocole de transport de courrier de bas niveau, généralement le bien nommé SMTP (*Simple Mail Transfer Protocol*).
- Elle peut exécuter un programme de courrier électronique externe en mode ligne de commande, si le système serveur dispose d'un tel programme.
- Elle peut utiliser l'API JavaMail, conçue pour supporter une gestion, un remplissage et un traitement complexes du courrier (voir à *http://java.sun.com/products/javamail*).
- Elle peut utiliser l'une des nombreuses classes disponibles gratuitement qui cachent les détails de l'envoi du courrier derrière des appels de méthodes simples et pratiques.

Pour l'envoi d'e-mail simple, nous vous recommandons la dernière approche du fait de sa simplicité. Pour des utilisations plus complexes, nous conseillons JavaMail, en particulier pour les servlets s'exécutant dans un serveur J2EE où les deux fichiers JAR dont JavaMail a besoin sont forcément disponibles.

Utiliser la classe MailMessage

Pour notre exemple, nous montrons une servlet qui se sert de la classe `com.oreilly.servlet.MailMessage`, non montrée ici mais disponible à *http://www.servlets.com*. Elle est modélisée d'après la classe `sun.net.smtp.SmtpClient` fournie par le JDK de Sun mais elle peut être utilisée sans les problèmes liés à l'emploi d'une classe de `sun.*` « non supportée et sujette aux modifications ». Elle propose aussi quelques améliorations particulièrement pratiques. Son utilisation est simple :

1. Appeler `MailMessage msg = new MailMessage()`. Vous pouvez aussi passer au constructeur le nom de la machine sur laquelle se trouve le serveur de courrier, en remplacement de la valeur `localhost` par défaut. La plupart des machines Unix peuvent agir en tant que serveur de courrier SMTP.
2. Appeler `msg.from(adresseEmetteur)`, en spécifiant l'adresse de l'émetteur. Il n'est pas nécessaire que l'adresse soit valide.
3. Appeler `msg.to(adresseDestinataire)`, en spécifiant l'adresse du destinataire. Cette méthode peut être appelée plusieurs fois s'il y a d'autres destinataires. Il existe aussi des méthodes `cc()` and `bcc()`.
4. Appeler `msg.setSubject(sujet)` pour fixer le sujet. Techniquement, ce n'est pas obligatoire, mais il est toujours bon de donner un sujet à son message.
5. Appeler `msg.setHeader(nom, valeur)` s'il faut définir d'autres en-têtes. Les en-têtes `To:`, `Cc:` et `Subject:` n'ont pas besoin d'être fixés car ils le sont automatiquement lors des appels aux méthodes `to()`, `cc()` et `setSubject()`. (La méthode `bcc()` ne fixe bien entendu aucun en-tête.) Vous pouvez employer `setHeader()` pour écraser les valeurs par défaut. Le nom et la valeur des en-têtes doivent suivre les règles données dans la RFC 822 à *http://www.ietf.org/rfc/rfc0822.txt*.
6. Appeler `PrintStream msg = msg.getPrintStream()` pour obtenir un flot de sortie pour le message.
7. Écrire le corps du message dans le `PrintStream`.

8. Appeler `msg.sendAndClose()` pour envoyer le message et fermer la connexion avec le serveur.

Données de formulaire dans un e-mail

L'*exemple 19-3* montre une servlet qui envoie à une liste de diffusion et par courrier électronique les données du formulaire qu'elle reçoit. Notez l'utilisation intensive de la classe `ParameterParser`.

Exemple 19-3. Envoyer un courrier à partir d'une servlet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.MailMessage;
import com.oreilly.servlet.ParameterParser;
import com.oreilly.servlet.ServletUtils;

public class MailServlet extends HttpServlet {

  static final String FROM = "MailServlet";
  static final String TO = "feedback-folks@attentive-company.com";

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    ParameterParser parser = new ParameterParser(req);
    String from = parser.getStringParameter("from", FROM);
    String to = parser.getStringParameter("to", TO);

    try {
      MailMessage msg = new MailMessage();  // On suppose localhost.
      msg.from(from);
      msg.to(to);
      msg.setSubject("Avis du client");

      PrintStream body = msg.getPrintStream();

      Enumeration enum = req.getParameterNames();
      while (enum.hasMoreElements()) {
        String name = (String)enum.nextElement();
        // Sauter to/from.
        if (name.equals("to") || name.equals("from")) continue;
        String value = parser.getStringParameter(name, null);
        body.println(name + " = " + value);
      }
```

Exemple 19-3. Envoyer un courrier à partir d'une servlet (suite)

```
      body.println();
      body.println("---");
      body.println("Envoyé par " + HttpUtils.getRequestURL(req));

      msg.sendAndClose();

      out.println("Merci de votre envoi...");
    }
    catch (IOException e) {
      out.println("Un problème est survenu pendant l'envoi...");
      log("Un problème est survenu pendant l'envoi de l'e-mail", e);
    }
  }
}
```

Cette servlet commence par déterminer les adresses « from » et « to » du message. Les valeurs par défaut sont mises dans les variables `FROM` et `TO`, bien qu'un formulaire puisse inclure des champs (probablement cachés) pour spécifier d'autres adresses. La servlet démarre ensuite un message e-mail SMTP. Elle se connecte à la machine locale et adresse le message. Puis, elle définit les en-têtes et remplit le corps du message avec les données du formulaire, en ignorant les variables `TO` et `FROM`. Enfin, elle envoie le message et remercie l'utilisateur de son envoi. S'il y a eu un problème, elle en informe l'utilisateur et journalise l'exception.

La classe `MailMessage` ne supporte pas encore les pièces jointes (même s'il pourrait être facilement ajouté). Pour des utilisations plus avancées, JavaMail est une bonne solution.

Utiliser les expressions régulières

Si vous êtes un programmeur de servlets ayant une expérience des scripts CGI en Perl et que vous êtes encore épris des possibilités des expressions régulières de Perl, alors cette section est pour vous. Nous montrons ici comment utiliser les expressions régulières en Java. Pour ceux d'entre vous qui ne sont pas familiers des expressions régulières, il s'agit d'un mécanisme autorisant une manipulation extrêmement avancée des chaînes de caractères avec un minimum de code. Les expressions régulières sont très bien expliquées dans le livre *Maîtrise des expressions régulières* de Jeffrey E. F. Friedl (Éditions O'Reilly).

Malgré toutes les classes et les possibilités que Sun a ajoutées au JDK au fil des ans, il manque encore un moteur d'expressions régulières[1]. Mais pas d'inquiétudes. Comme avec la plupart des fonctionnalités de Java, ce que vous ne pouvez obtenir de Sun, un autre fournisseur vous le vendra probablement pour un prix raisonnable, ou quand il s'agit de quelque chose d'aussi utilisé que les expressions régulières, vous le trouverez certainement sous forme de bibliothèque en *open source*. En fait, il existe bel et bien un moteur d'expressions régulières disponibles au sein du projet Jakarta d'Apache. Il a été conçu à l'origine par Jonathan Locke et il est disponible sous la licence Apache. Cette licence est bien plus permissive que la GNU General Public License (GPL) car elle per-

1. *N.d.T.* : le JDK 1.4 contient maintenant un moteur d'expressions régulières.

met aux développeurs d'utiliser un moteur d'expressions régulières pour créer de nouveaux produits sans leur imposer de les sortir en version en *open source*.

Trouver les liens à l'aide des expressions régulières

Pour montrer l'utilisation des expressions régulières, utilisons le moteur d'Apache pour écrire une servlet qui extrait et affiche dans une liste tous les liens HTML <A HREF> trouvés dans une page. Le code est donné à *exemple 19-4*.

Exemple 19-4. Rechercher tous les liens

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.*;

import org.apache.regexp.*;

public class Links extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    // L'URL à traiter est donnée dans l'information de chemin supplémentaire.
    // http://localhost:8080/servlet/Links/http://www.servlets.com/
    String url = req.getPathInfo();
    if (url == null || url.length() == 0) {
      res.sendError(res.SC_BAD_REQUEST, "Veuillez donner l'URL à lire " +
                    "dans l'information de chemin supplémentaire");
      return;
    }
    url = url.substring(1);  // Supprimer le dernier '/'.

    String page = null;
    try {
      // Demander la page.
      HttpMessage msg = new HttpMessage(new URL(url));
      BufferedReader in =
        new BufferedReader(new InputStreamReader(msg.sendGetMessage()));

      // Lire la réponse complète dans un String.
      StringBuffer buf = new StringBuffer(10240);
      char[] chars = new char[10240];
      int charsRead = 0;
      while ((charsRead = in.read(chars, 0, chars.length)) != -1) {
```

Exemple 19-4. Rechercher tous les liens (suite)

```
        buf.append(chars, 0, charsRead);
      }
      page = buf.toString();
    }
    catch (IOException e) {
      res.sendError(res.SC_NOT_FOUND,
                    "L'extracteur de liens ne peut lire à partir de " + url +
                    " :<BR>" + ServletUtils.getStackTraceAsString(e));
      return;
    }

    out.println("<HTML><HEAD><TITLE>Extracteur de liens</TITLE>");

    try {
      // Nous devons donner une <BASE> afin que les liens relatifs
      // fonctionnent correctement. Si la page en inclus déjà une, nous
      // pouvons l'utiliser.
      RE re = new RE("<base[^>]*>", RE.MATCH_CASEINDEPENDENT);
      boolean hasBase = re.match(page);

      if (hasBase) {
        // Utiliser la <BASE> existante.
        out.println(re.getParen(0));
      }
      else {
        // Calculer la base à partir de l'URL, utiliser tout jusqu'au '/'
        // final.
        re = new RE("http://.*/", RE.MATCH_CASEINDEPENDENT);
        boolean extractedBase = re.match(url);
        if (extractedBase) {
          // OK? afficher la base calculée.
          out.println("<BASE HREF=\"" + re.getParen(0) + "\">");
        }
        else {
          // Pas de barre oblique finale, en ajouter une.
          out.println("<BASE HREF=\"" + url + "/" + "\">");
        }
      }

      out.println("</HEAD><BODY>");

      out.println("Les liens de <A HREF=\"" + url + "\">" + url + "</A>" +
                  " sont : <BR>");
      out.println("<UL>");

      String search = "<a\\s+[^<]*</a\\s*>";
      re = new RE(search, RE.MATCH_CASEINDEPENDENT);
```

Exemple 19-4. Rechercher tous les liens (suite)

```
      int index = 0;
      while (re.match(page, index)) {
        String match = re.getParen(0);
        index = re.getParenEnd(0);
        out.println("<LI>" + match + "<BR>");
      }

      out.println("</UL>");
      out.println("</BODY></HTML>");
    }
    catch (RESyntaxException e) {
      // Ne doit jamais arriver car les chaînes de recherche sont figées
      // dans le code.
      e.printStackTrace(out);
    }
  }
}
```

Une copie d'écran est donnée à la *figure 19-1*.

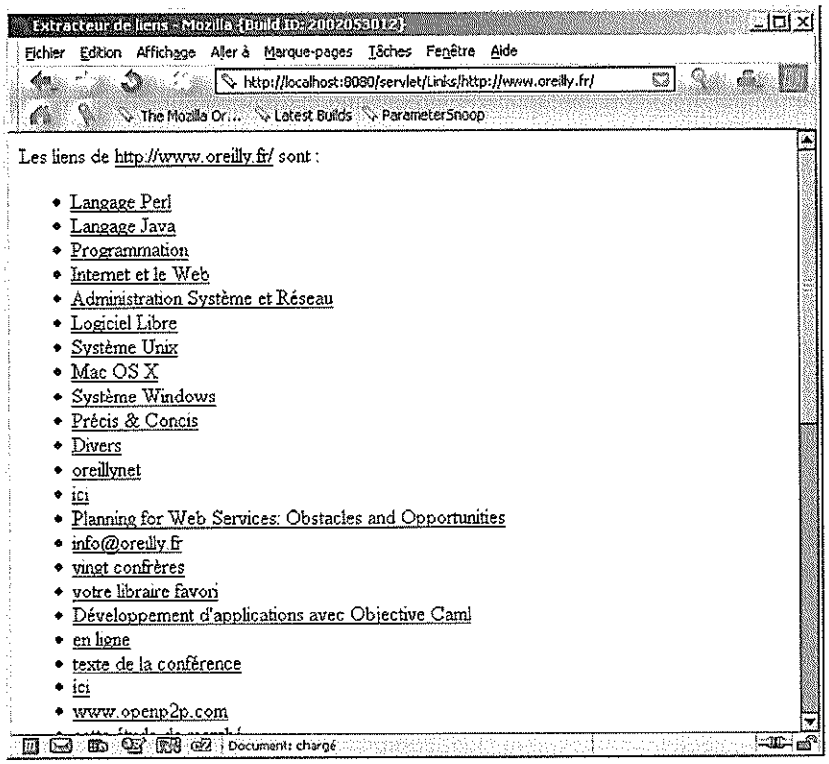


Figure 19-1. Navigation avec bande passante très faible

Examinons le code. La servlet commence par déterminer l'URL de laquelle les liens doivent être extraits en analysant son information de chemin supplémentaire. Cela signifie que cette servlet doit être invoquée ainsi : *http://localhost:8080/servlet/Links/http://www.servlets.com*. Puis, la servlet lit le contenu de cette URL avec la classe `HttpMessage` qu'elle stocke dans un `String`. Pour les pages très grandes, cette approche n'est pas efficace, mais c'est suffisant pour un exemple dans un livre.

L'étape suivante consiste à s'assurer que la page de sortie a la bonne balise `<BASE>` afin que les liens relatifs de la liste soient interprétés correctement par le navigateur. S'il existe déjà une balise `<BASE>` sur la page d'entrée, nous pouvons l'utiliser. Nous la recherchons donc avec l'expression régulière `<base[^>]*>`. Nous passons cette chaîne au constructeur de `org.apache.regexp.RE` ainsi qu'un drapeau servant à la casse. Cette syntaxe est standard et exactement la même qu'en Perl. Elle demande de rechercher le texte `<base`, suivi d'un nombre quelconque de caractères autres que `>`, suivis du texte `>`. Si une correspondance est trouvée, nous l'extrayons par l'appel `re.getParen(0)` qui récupère la correspondance la plus extérieure (les correspondances peuvent être imbriquées dans des parenthèses).

Si la page ne contient pas de balise `<BASE>`, nous devons en construire une. `<BASE>` doit être tout ce qui se trouve dans l'URL source jusqu'à la barre oblique finale comprise.[2] Nous pouvons extraire cette information à l'aide de l'expression régulière simple `http://.*/`. Elle indique de trouver le texte `http://`, suivi d'un nombre quelconque de caractères, suivis par `/`. Le motif `.*` lit autant de caractères que possible tout en satisfaisant le reste de la condition (ce que l'on appelle être *gourmand*) afin que cette expression retourne tous les caractères jusqu'à la barre oblique finale incluse. Si elle n'existe pas, nous l'ajoutons simplement.

Enfin, nous extrayons les balises `<A HREF>` de la page à l'aide de l'expression régulière plutôt complexe `<a\s+[^<]*</a\s*>`. (Vous noterez que dans le code, nous avons appliqué l'échappement aux caractères `\` par un caractère `\` supplémentaire.) Cette expression indique de rechercher le texte `<a`, suivi d'un ou de plusieurs espaces (`\s` signifie espace), suivis d'un nombre quelconque de caractères autres que `<`, suivis du texte `</a`, suivi d'un nombre quelconque d'espaces, suivis de `>`. En fait, on extrait les balises `<A HREF>` à partir du `<A` de début jusqu'au `</A>` final, le tout sans tenir compte de la casse, des espaces inutiles et en s'assurant que l'on ne sélectionne pas par erreurs les balises comme `<APPLET>`. Chaque correspondance trouvée est affichée dans la liste et la recherche continue avec un indice permettant d'enregistrer le prochain point de départ.

Pour plus d'informations sur ce qui peut être fait avec les expressions régulières en Java, consultez la documentation qui vient avec la bibliothèque.

Exécuter des programmes

Parfois, une servlet doit exécuter un programme externe. En général, c'est important dans les situations où un programme externe offre des fonctionnalités non facilement

2. Avec un logique supplémentaire non montrée ici (impliquant une préconnexion au serveur), cette approche peut échouer pour des URL comme *http://www.jdom.org/news* qui redirige vers *http://www.jdom.org/news/*. Pour être sûr du fonctionnement, mettez une barre oblique finale dans l'URL passée à cette servlet.

disponibles en Java. Par exemple, une servlet pourrait appeler un programme externe pour effectuer la manipulation d'une image ou pour contrôler l'état du serveur. Le lancement d'un programme externe soulève des problèmes de sécurité. C'est pourquoi cette action ne peut être effectuée que par des servlets s'exécutant avec un gestionnaire de sécurité tolérant, plus précisément un gestionnaire de sécurité qui accorde à la servlet la permission d'appeler la méthode exec() de `java.lang.Runtime`.

Finger

Le programme *finger* interroge un ordinateur (peut-être distant) afin d'obtenir une liste des utilisateurs actuellement connectés. Il est disponible sur tous les système Unix et sur quelques machines Windows NT ayant des fonctions réseau. Le programme *finger* fonctionne en se connectant à un démon *finger* (habituellement appelé *fingerd*) qui écoute sur le port 79. *finger* fait sa requête sur *fingerd* en utilisant un protocole « finger » spécifique et *fingerd* répond avec les informations appropriées. La plupart des systèmes Unix exécutent *fingerd*, bien que beaucoup d'administrateurs soucieux de garantir une certaine sécurité le désactivent afin de limiter les informations pouvant servir à pénétrer leur système. Il est assez rare de trouver *fingerd* sur un système Windows. Exécuté sans arguments, *finger* rapporte la liste de tous les utilisateurs de la machine locale. *fingerd* doit tourner sur la machine locale. Voici un exemple :

```
% finger
Login       Name                 TTY Idle When       Office
jhunter     Jason Hunter         q0  3:13 Thu 12:13
kbautista   Kathlyn Bautista     q1       Thu 12:18
```

En passant en argument un nom d'utilisateur, *finger* retourne les informations sur cet utilisateur uniquement :

```
% finger jhunter
Login name: jhunter                     In real life: Jason Hunter
Directory: /usr/people/jhunter          Shell: /bin/tcsh
On since Jan  1 12:13:28 on ttyq0 from :0.0
3 hours 13 minutes Idle Time
On since Jan  1 12:13:30 on ttyq2 from :0.0
```

En passant en argument le nom d'une machine, *finger* donne la liste de tous les utilisateurs présents sur la machine spécifiée. *fingerd* doit tourner sur la machine distante :

```
% finger @deimos
Login    Name                 TTY Idle When       Office
bday     Bill Day             q0   17d Mon 10:45
```

Et, bien sûr, en donnant un nom d'utilisateur et un nom de machine, *finger* retourne les informations concernant cet utilisateur sur cette machine :

```
% finger bday@deimos
[deimos.engr.sgi.com]
Login name: bday                        In real life: Bill Day
Directory: /usr/people/bday             Shell: /bin/tcsh
On since Dec 15 10:45:22 on ttyq0 from :0.0
17 days Idle Time
```

Exécuter la commande finger

Supposons qu'une servlet veuille accéder aux informations retournées par *finger*. Elle a deux options : elle peut établir une connexion par socket avec *fingerd* et demander les informations comme le ferait tout client *finger*, ou elle peut exécuter le programme *finger* en ligne de commande pour effectuer la connexion à sa place et lire les informations de sa sortie. Nous allons montrer cette seconde technique.

L'*exemple 19-5* montre comment une servlet peut exécuter la commande `finger` pour voir qui est connecté sur la machine locale. Elle lit la sortie de la commande et l'affiche avec son flot de sortie.

Exemple 19-5. Exécuter la commande finger à partir d'une servlet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class Finger extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    String command = "finger";

    Runtime runtime = Runtime.getRuntime();
    Process process = null;
    try {
      process = runtime.exec(command);
      BufferedReader in =
        new BufferedReader(new InputStreamReader(process.getInputStream()));

      // Lire et afficher la sortie.
      String line = null;
      while ((line = in.readLine()) != null) {
        out.println(line);
      }
    }
    catch (Exception e) {
      out.println("Problème avec finger : " +
                  ServletUtils.getStackTraceAsString(e));
    }
  }
}
```

Cette servlet utilise la commande exec() comme pourrait le faire toute autre classe Java. Elle exécute la commande *finger*, puis elle lit et affiche la sortie. S'il y a un problème, la servlet intercepte une exception et affiche la trace de la pile à l'utilisateur. Cette servlet suppose que la commande *finger* existe dans le chemin de recherche par défaut. Si ce n'est pas le cas, changez la chaîne de commande pour indiquer le chemin où se trouve *finger*.

Nous devons souligner que, bien que Java exécute un code natif lorsqu'il exécute le programme *finger*, il ne s'expose pas aux risques existant normalement lorsque l'on exécute du code natif. La raison en est que le programme *finger* s'exécute dans un processus séparé. Il peut se planter ou être détruit sans conséquence pour le serveur exécutant la servlet.

Exécuter finger avec des arguments

Supposons maintenant que nous voulions passer un argument à la commande *finger*. C'est un peu différent. exec() prend, soit une seule chaîne qui spécifie la commande, soit un tableau de chaînes qui spécifie une commande et les arguments à passer à cette commande. Pour exécuter *finger jhunter*, le code ressemblerait à celui de l'*exemple 19-6*.

Exemple 19-6. Ajouter un paramètre à la commande exécutée

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.ServletUtils;

public class Finger extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    String[] command = { "finger", "jhunter" };  // Seul changement !

    Runtime runtime = Runtime.getRuntime();
    Process process = null;
    try {
      process = runtime.exec(command);
      BufferedReader in =
        new BufferedReader(new InputStreamReader(process.getInputStream()));

      // Lire et afficher la sortie.
      String line = null;
      while ((line = in.readLine()) != null) {
        out.println(line);
      }
    } catch (Exception e) {
```

Exemple 19-6. Ajouter un paramètre à la commande exécutée (suite)

```
      out.println("Problème avec finger : " +
                  ServletUtils.getStackTraceAsString(e));
    }
  }
}
```

La variable `command` est maintenant le tableau de chaînes `{"finger", "jhunter"}`. La commande ne fonctionnerait pas avec la seule chaîne `"finger jhunter"`.

Exécuter Finger avec redirection de la sortie

Enfin, supposons que nous voulions rediriger la sortie de notre commande *finger*. On pourrait rediriger la sortie vers un fichier pour une utilisation ultérieure, comme dans `finger jhunter > /tmp/jhunter`. Ou nous pourrions la rediriger vers le programme *grep* pour supprimer toutes les références à un certain utilisateur, comme dans `finger | grep -v jhunter`.

Cette tâche est plus difficile qu'il n'y paraît. Si la variable `command` est mise à `finger | grep -v jhunter`, Java traite cette chaîne comme le nom d'un seul programme, qu'il est sûr de ne pas trouver. Si la variable `command` est le tableau de chaînes `{"finger", "|", "grep", "-v", "jhunter"}`, Java exécute la commande *finger* et lui passe les quatre autres chaînes en argument, ce qui risque fort de perturber *finger*.

La solution demande de comprendre que la redirection est une fonctionnalité du *shell*. Le shell est le programme dans lequel vous entrez normalement les commandes. Sous Unix, les shells les plus répandus sont *csh*, *tcsh*, *bash* et *sh*. Sous Windows 95/98, le shell est habituellement *command.com*. Sous Windows NT et Windows 2000, le shell est, soit *command.com*, soit *cmd.exe*.

Au lieu d'exécuter *finger* directement, nous pouvons lancer un shell et lui indiquer la chaîne de commandes que nous voulons exécuter. Cette chaîne peut contenir la commande *finger* et toutes sortes de redirections. Le shell analyse la commande, reconnaît et effectue correctement la redirection. La commande exacte nécessaire à l'exécution d'un shell et d'un programme dépend du shell et donc du système d'exploitation. Cette technique limite donc l'indépendance de la plate-forme des servlets qui l'utilisent. Sur un système Unix, la variable `command` suivante demande à *csh* d'exécuter la commande `finger | grep -v jhunter` :

```
String[] command = { "/bin/csh", "-c", "finger | grep -v jhunter" };
```

Le programme Java exécute */bin/csh*. Deux arguments sont passés à *csh* : `-c`, qui lui demande d'exécuter le paramètre suivant, et `finger | grep -v jhunter`, qui est exécuté par le shell.

Sur un système Windows, la variable `command` ressemble à cela :

```
String[] command = { "command.com", "/c", "finger | grep -v jhunter" };
```

L'argument `/c` à *command.com* fonctionne comme `-c` pour *csh* et, oui, le suffixe *.com* est indispensable. Pour Windows NT, il faut noter que l'utilisation de *cmd.exe* pose un problème car il redirige sa sortie vers une nouvelle fenêtre au lieu de l'exécutif Java qui l'a lancé. En fait, même lancer votre serveur à partir d'un shell *cmd.exe* peut conduire à l'échec de la commande *command.com*.

Utiliser les méthodes natives

Même si Sun milite pour le 100% Pure Java, il y a encore de la place pour du code natif. Vous avez besoin du code natif pour réaliser des choses que Java (et les programmes externes lancés par Java) ne peut pas faire : verrouiller les fichiers, accéder aux identifiants des utilisateurs, accéder à la mémoire partagée, envoyer des fax, etc. Le code natif est aussi utile pour accéder à des données centrales au travers de passerelles non-Java. Enfin, dans des situations où chaque petit gain de performance peut être vital, les bibliothèques en code natif peuvent donner un sérieux coup de fouet au serveur.

Cependant, le code natif ne devrait être utilisé qu'en cas d'absolue nécessité, puisque si le code natif d'une servlet va dans une certaine direction, le serveur entier y va ! Les protections de Java ne peuvent protéger le serveur des dysfonctionnements du code natif. C'est pourquoi il est n'est pas recommandé d'utiliser le pont JDBC-ODBC natif à partir d'une servlet car la plupart des pilotes ODBC semblent avoir des problèmes avec les accès multithread. Le code natif limite aussi l'indépendance de la plate-forme d'une servlet. Bien que cela puisse ne pas être très important pour des servlets personnalisées liées à un serveur particulier, il ne faut pas l'oublier.

La façon d'accéder à des méthodes natives dépend du serveur web et de la JVM utilisée. Pour parler de façon générale, disons que vous pouvez supposer que votre serveur web et la JVM supporte JNI (*Java Native Interface*). L'utilisation de JNI est assez compliquée et même une introduction dépasse le cadre de ce chapitre.

Voici ce qu'il faut retenir lorsque l'on utilise JNI avec les servlets :

- Seuls les gestionnaires de sécurité du serveur les plus libéraux autorisent une servlet à exécuter du code natif.
- Il y a un bogue dans le JDK 1.1.x qui interdit le chargement du code natif par une classe qui a été chargée avec un chargeur de classes personnalisé (comme le chargeur de classes qui charge les servlets à partir de leur répertoire par défaut). Les servlets utilisant du code natif doivent donc se trouver dans le chemin des classes du serveur (*serveur_root/classes*).
- Le répertoire où se trouve la bibliothèque partagée (ou la bibliothèque chargée dynamiquement ou DLL) qui contient le code natif dépend du serveur web et de la JVM. Certains serveurs recherchent les bibliothèques partagées à des endroits spécifiques. Si le serveur ne fournit pas un répertoire spécifique pour les bibliothèques partagées, essayez de les placer dans des endroits spécifiques à la JVM, comme *racine_jdk\bin* ou sous *racine_jdk/lib* (où *racine_jdk* est la racine de l'installation du JDK), ou essayez un endroit spécifique au système d'exploitation comme *racine_windows\system32* ou */usr/lib*.

Agir comme un client RMI

Au chapitre 10, *Communication applet-servlet*, nous avons vu comment une servlet peut fonctionner en tant que serveur RMI. Ici, nous allons voir comment une servlet fonctionne en tant que client RMI. En prenant le rôle d'un client RMI, une servlet peut exploiter les services des autres serveurs pour accomplir sa tâche, coordonner ses efforts à ceux des autres serveurs ou servlets sur ces serveurs, et/ou agir comme un mandataire pour des applets qui ne peuvent communiquer elles-mêmes avec les serveurs RMI.

L'*exemple 19-7* montre DaytimeClientServlet, une servlet qui récupère l'heure courante du jour à partir du serveur RMI DaytimeServlet montré au chapitre 10.

Exemple 19-7. Servlet en tant que client RMI

```
import java.io.*;
import java.rmi.*;
import java.rmi.registry.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DaytimeClientServlet extends HttpServlet {

  DaytimeServer daytime;

  // Retourne une référence au DaytimeServer ou null en cas de problème.
  protected DaytimeServer getDaytimeServer() {
    // Si vous utilisez la fonctionnalité de chargement de code dynamique
    // de RMI, il faut définir un gestionnaire de sécurité comme
    // RMISecurityManager.
    //if (System.getSecurityManager() == null) {
    //  System.setSecurityManager(new RMISecurityManager());
    //}

    try {
      Registry registry =
        LocateRegistry.getRegistry(getRegistryHost(), getRegistryPort());
      return (DaytimeServer)registry.lookup(getRegistryName());
    }
    catch (Exception e) {
      log("Problème pour obtenir une référence à DaytimeServer", e);
      return null;
    }
  }

  private String getRegistryName() {
    String name = getInitParameter("registryName");
    return (name == null ? "DaytimeServlet" : name);
  }

  private String getRegistryHost() {
    // Retourner le nom de machine donné par "registryHost" ou
    // null pour indiquer localhost si le nom n'a pas été donné.
    return getInitParameter("registryHost");
  }

  private int getRegistryPort() {
    try { return Integer.parseInt(getInitParameter("registryPort")); }
    catch (NumberFormatException e) { return Registry.REGISTRY_PORT; }
  }
```

Exemple 19-7. Servlet en tant que client RMI (suite)

```
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();

    // Obtenir un objet daytime si ce n'est déjà fait.
    if (daytime == null) {
      daytime = getDaytimeServer();
      if (daytime == null) {
        // Impossible de l'obtenir, indiquer notre indisponibilité.
        throw new UnavailableException("Impossible de localiser daytime");
      }
    }

    // Obtenir et affciher l'heure courante sur la machine (distante).
    out.println(daytime.getDate().toString());
  }
}
```

Cette servlet devrait vous rappeler l'applet vue au chapitre 10. Les servlets et les applets effectuent les mêmes étapes de base pour accéder à un serveur RMI. Elles localisent un registre en utilisant un nom de machine et un numéro de port et l'utilisent ensuite pour obtenir une référence à l'objet distant. La seule différence importante est qu'une servlet, si elle exploite la fonctionnalité RMI de chargement de code dynamique pour télécharger automatiquement les fichier de classe du stub à partir d'une autre machine, doit commencer par vérifier qu'elle s'exécute sous la protection d'un gestionnaire de sécurité afin de se protéger du stub téléchargé potentiellement dangereux. Une applet étant certaine de s'exécuter sous le contrôle d'un gestionnaire de sécurité pour applets, cette étape n'est pas nécessaire. Cependant, une servlet peut fonctionner sans gestionnaire de sécurité par défaut. Il lui en faut donc un avant d'agir comme client RMI.

Débogage

La phase de test/débogage est l'un des aspects les plus difficiles du développement des servlets. Elles ont tendance à impliquer beaucoup d'interaction client/serveur, facilitant les erreurs, mais les rendant difficiles à reproduire. Il peut aussi être difficile de traquer la cause d'erreurs non évidentes car les servlets ne fonctionnent pas bien avec les débogueurs standards puisqu'elles s'exécutent dans un serveur web multithread et généralement complexe. Voici quelques conseils et suggestions pour vous aider.

Examiner les journaux

Quand vous pensez qu'il peut y avoir un problème, vérifiez dans les journaux. La plupart des serveurs sortent un journal d'erreurs, où vous pouvez trouver une liste de toutes les erreurs observées par le serveur, et un journal d'événements, où vous pouvez trouver une liste des événements des servlets intéressants. Le journal des événements peut même contenir des messages journalisés par les servlets *via* la méthode `log()`, mais pas toujours.

Notez que beaucoup de serveurs utilisent un tampon pour la sortie vers ces journaux afin d'améliorer les performances. Lorsque vous pourchassez un problème, vous devez désactiver cette mise en tampon (habituellement en réduisant la taille du tampon à 0 octet), afin de voir les problèmes quand ils surgissent. Assurez-vous de remettre la taille du tampon à une valeur raisonnable par la suite.

Sortir des informations supplémentaires

Si vous ne voyez pas une indication du problème dans les journaux du serveur, essayez de faire en sorte que votre servlet journalise des informations supplémentaires avec la méthode `log()`. Comme vous l'avez vu dans les exemples de ce livre, nous journalisons généralement les traces de la pile et d'autres cas d'erreurs. Pendant le débogage, vous pouvez ajouter quelques commandes `log()` temporaires pour avoir une idée générale de l'exécution de votre code et de la valeur des variables de la servlet. Quelquefois, il est pratique d'entourer les commandes `log()` par des clauses `if` afin de les rendre actives en mettant à `true` un paramètre d'initialisation.

L'extraction des informations supplémentaires des journaux du serveur peut à la longue s'avérer lourde. Pour que l'information de débogage temporaire soit facile à trouver, la servlet peut envoyer ses informations au client (via le `PrintWriter`) ou à une console sur le serveur (via `System.out`). Tous les serveurs n'associent pas une console à `System.out` d'une servlet ; certains redirigent la sortie vers un fichier.

Utiliser un débogueur standard

Il est aussi possible d'utiliser un débogueur standard pour traquer les problèmes d'une servlet, même si la façon de le faire peut ne pas être évidente. Après tout, vous ne pouvez pas déboguer une servlet directement car les servlets ne sont pas des programmes autonomes. Les servlets sont des extensions du serveur et, en tant que telles, elles doivent s'exécuter dans le serveur.

Heureusement, Tomcat est un serveur web pur Java bien adapté au débogage des servlets. La seule astuce est de démarrer Tomcat à partir du débogueur. La procédure exacte peut varier selon la version de Tomcat (ou de tout autre serveur web basé sur Java), mais les grandes lignes sont généralement identiques :

1. Définissez le chemin de classes de votre débogueur afin qu'il contienne les classes et les fichiers JAR nécessaires à l'exécution de Tomcat. Vous pouvez aussi examiner la sortie affichée au démarrage et les scripts de lancement du serveur (*tomcat.sh* et *tomcat.bat*) pour déterminer ce chemin de classes.
2. Définissez le chemin de classes de votre débogueur afin qu'il contienne aussi les classes des servlets et les classes utilitaires, typiquement le répertoire *WEB-INF/classes* et les fichiers dans *WEB-INF/lib*. En général, vous ne voulez pas de ces répertoires et de ces fichiers JAR dans votre chemin de classes car cela désactive le rechargement de servlets. Cependant, cette inclusion est utile pour le débogage. Elle permet à votre débogueur de placer des points d'arrêt dans une servlet avant que le chargeur de servlets personnalisé responsable du chargement des classes de *WEB-INF* ne charge la servlet.
3. Une fois que le chemin de classes est correct, démarrez le débogage du serveur en lançant le classe du serveur contenant la méthode `main()`. Pour Tomcat 3.2, la classe est `org.apache.tomcat.startup.Tomcat`. Les autres serveurs basés sur Java et les

futures versions de Tomcat peuvent utiliser une classe différente. Consultez le script de démarrage pour déterminer le nom de la classe principale.

4. Tomcat peut se plaindre que des propriétés système ou des variables d'environnement ne sont pas définies. Par exemple, Tomcat 3.2 examine la propriété système `tomcat.home` ou la variable d'environnement `TOMCAT_HOME` afin de déterminer son répertoire de base. Donnez leur les bonnes valeurs.
5. Placez des points d'arrêt là où vous le souhaitez dans votre servlet, utilisez un navigateur web pour effectuer une requête sur la servlet (*http://localhost:8080/servlet/ServletToDebug*). Vous devriez voir l'exécution s'arrêter sur vos points d'arrêt.

Beaucoup de débogueurs d'IDE cachent ces détails et autorise le débogage à l'aide d'un serveur intégré exécutant les servlets. Cependant, il est souvent bon de connaître la procédure manuelle car les serveurs fournis avec les débogueurs d'IDE sont souvent des anciennes versions.

Certains conteneurs de servlets (ceux qui normalement ne fonctionnent qu'avec un serveur web non-Java) ont des versions pures Java autonomes créées spécifiquement pour le débogage. L'avantage d'utiliser ces serveurs, quand ils sont disponibles, est qu'il est facile de passer les fichiers de configuration personnalisée du serveur web de l'environnement de production à l'environnement de test, et vice versa.

Examiner la requête du client

Parfois, quand une servlet ne se comporte pas comme espéré, il est utile d'examiner la requête HTTP à laquelle elle répond. Si vous êtes familier de la structure de HTTP, vous pouvez lire la requête et voir exactement ce qui pourrait troubler une servlet[3]. Une façon de voir la requête consiste à remplacer le processus du serveur web par une application serveur personnalisée qui affiche tout ce qu'elle reçoit. L'*exemple 19-8* montre un tel serveur.

Exemple 19-8. Intercepter la requête d'un client

```
import java.io.*;
import java.net.*;
import java.util.*;

public class SocketWatch {

  private static void printUsage() {
    System.out.println("utilisation : java SocketWatch port");
  }

  public static void main(String[] args) {
    if (args.length < 1) {
      printUsage();
```

3. Bien sûr, si vous n'êtes pas familier de la structure de HTTP, c'est vous qui risquez d'être troublé. Dans ce cas, nous vous recommandons de lire l'introduction à HTTP du chapitre 2, *Introduction aux servlets HTTP*, et le livre *HTTP Pocket Reference* de Clinton Wong (O'Reilly & Associates).

Exemple 19-8. Intercepter la requête d'un client (suite)

```
      return;
    }

    // Le premier paramètre est le port sur lequel écouter.
    int port;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (NumberFormatException e) {
      printUsage();
      return;
    }

    try {
      // Établir une socket serveur pour accepter les connexions des clients.
      // Quand une connexion survient, la passer au thread de gestion.
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        Socket request = ss.accept();
        new HandlerThread(request).start();
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}

class HandlerThread extends Thread {

  Socket s;

  public HandlerThread(Socket s) {
    this.s = s;
  }

  public void run() {
    try {
      // Afficher chaque octet dès qu'il arrive sur la socket.
      InputStream in = s.getInputStream();
      byte[] bytes = new byte[1];
      while ((in.read(bytes)) != -1) {
        System.out.print((char)bytes[0]);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Lancez ce serveur écoutant sur le port 8080 en entrant la commande suivante :

```
java SocketWatch 8080
```

Notez que deux applications ne peuvent écouter en même temps sur la même socket. Vous devez donc commencer par vous assurer qu'aucune autre application n'écoute sur le port choisi. Une fois que votre serveur s'exécute, vous pouvez lancer des requêtes HTTP comme s'il s'agissait d'un serveur web normal. Par exemple, vous pouvez utiliser un navigateur web pour surfer sur *http://localhost:8080*. Lorsque SocketWatch reçoit la requête HTTP du navigateur, elle l'envoie sur sa sortie standard pour que vous l'examiniez. Le navigateur va peut-être se mettre en attente d'une réponse qui ne viendra jamais. Vous pouvez arrêter cette attente en appuyant sur le bouton Stop.

Voici un exemple de sortie de SocketWatch qui montre les détails d'une requête GET faite sur *http://localhost:8080* :

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; IRIX 6.2 IP22)
Pragma: no-cache
Host: localhost:8080
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Cookie: JSESSIONIN=To1010mC10934500694587412At
```

Créer une requête client personnalisée

En plus d'intercepter et d'examiner une requête HTTP d'un client, vous pouvez trouver utile de créer votre propre requête HTTP. Pour cela, il faut vous connecter à la socket sur laquelle le serveur écoute, puis entrer manuellement une requête HTTP correctement structurée. Pour établir la connexion, vous pouvez utiliser le programme *telnet*, disponible sur toutes les machines Unix et sur la plupart des machines Windows en réseau. *telnet* accepte en argument la machine et le numéro de port auxquels il doit se connecter. Une fois la connexion établie, vous pouvez faire une requête qui ressemble à ce que vous avez vu à la section précédente. Heureusement, votre requête peut être bien plus simple, il suffit de spécifier la première ligne, indiquant ce que vous voulez, et la dernière ligne, qui doit être une ligne vide indiquant la fin de la requête. Par exemple :

```
% telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /servlet/ParameterSnoop?name=value HTTP/1.0

HTTP/1.1 200 OK
Server: Tomcat Web Server/3.2
Content-Type: text/plain
Connection: close
Date: Sun, 25 Jun 2000 20:29:06 GMT

Query String:
name=value
```

```
Request Parameters:
name (0): value
Connection closed by foreign host.
```

Comme c'est trop souvent le cas, Windows se comporte un peu différemment de ce que vous espériez. Le programme *telnet.exe* par défaut de Windows 95/98/NT formate mal la plupart des réponses des serveurs web car il ne comprend pas que sur le web un saut de ligne doit être traité comme un saut de ligne et un retour chariot. À la place de *telnet.exe*, les programmeurs Windows peuvent utiliser le programme Java montré à l'*exemple 19-9*.

Exemple 19-9. Une autre manière de se connecter à un serveur web

```
import java.io.*;
import java.net.*;
import java.util.*;

public class HttpClient {

  private static void printUsage() {
    System.out.println("utilisation : java HttpClient host port");
  }

  public static void main(String[] args) {
    if (args.length < 2) {
      printUsage();
      return;
    }

    // Host est le premier paramètre, port est le second.
    String host = args[0];
    int port;
    try {
      port = Integer.parseInt(args[1]);
    }
    catch (NumberFormatException e) {
      printUsage();
      return;
    }

    try {
      // Ouvrir une socket sur le serveur.
      Socket s = new Socket(host, port);

      // Démarrer un thread pour envoyer l'entrée clavier au serveur.
      new KeyboardInputManager(System.in, s).start();

      // Maintenant afficher tout ce que l'on reçoit de la socket.
      BufferedReader in =
        new BufferedReader(new InputStreamReader(s.getInputStream()));
      String line;
      while ((line = in.readLine()) != null) {
```

Exemple 19-9. Une autre manière de se connecter à un serveur web (suite)

```
        System.out.println(line);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}

class KeyboardInputManager extends Thread {

  InputStream in;
  Socket s;

  public KeyboardInputManager(InputStream in, Socket s) {
    this.in = in;
    this.s = s;
    setPriority(MIN_PRIORITY);  // Priorité plus élevée pour la lecture sur la
                                // socket. Je voudrais utiliser un select() !
    setDaemon(true);  // Laisser l'application mourir même lorsque le thread
                      // s'exécute.
  }

  public void run() {
    try {
      BufferedReader keyb = new BufferedReader(new InputStreamReader(in));
      PrintWriter server = new PrintWriter(s.getOutputStream());

      String line;
      System.out.println("Connecté... Entrez votre requête HTTP manuelle");
      System.out.println("-------------------------------------------");
      while ((line = keyb.readLine()) != null) {
        server.print(line);
        server.print("\r\n");  // Les lignes HTTP se terminent par \r\n.
        server.flush();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Ce programme `HttpClient` fonctionne de façon similaire à *telnet* :

```
% java HttpClient localhost 8080
Connecté... Entrez votre requête HTTP manuelle
-------------------------------------------
GET /index.html HTTP/1.0

HTTP/1.0 200 OK
Content-Type: text/html
```

```
Content-Length: 2582
Last-Modified: Fri, 15 Sep 2000 22:20:15 GMT
Servlet-Engine: Tomcat Web Server/3.2 (JSP 1.1; Servlet 2.2; Java 1.2.2;
Windows NT 4.0 x86; java.vendor=Sun Microsystems Inc.)

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <title>Tomcat v3.2</title>
</head>
...
```

Utiliser un outil tiers

Les outils provenant de tierce-parties promettent d'apporter de nouvelles possibilités et une facilité d'utilisation dans le débogage de servlets. IBM AlphaWorks produit un programme appelé Distributed Application Tester (DAT) qui examine les requêtes et les réponses HTTP et HTTPS, permettant ainsi de visualiser et d'enregistrer les deux côtés d'un échange client/serveur. DAT inclut la possibilité de tester la fonctionnalité et les performances de l'application web en générant automatiquement des requêtes et en analysant les réponses. Le programme est en pur Java mais il vient avec un outil d'installation qui ne fonctionne que sur Windows. Sa seule licence est une évaluation pendant 90 jours car il est en version alpha et ce curieusement depuis janvier 1999. DAT est disponible à *http://www.alphaworks.ibm.com*.

Allaire, le créateur du plug-in populaire JRun (après leur rachat de LiveSoftware), dispose d'un outil de débogage des servlets peu connu, appelé ServletDebugger. Il est conçu pour aider à tester et déboguer une servlet par programmation. ServletDebugger n'a pas besoin d'un serveur web ou d'un navigateur pour effectuer une requête. À la place, vous utilisez un ensemble de classes pour écrire une petite classe stub qui prépare et exécute une requête. Le stub spécifie tout : les paramètres d'initialisation de la servlet, les en-têtes HTTP de la requête et les paramètres de la requête. ServletDebugger est facile à utiliser et bien adapté aux tests automatiques. Son plus gros inconvénient est qu'il demande un effort supplémentaire pour bien préparer une requête réaliste. ServletDebugger est caché bien profond dans la grille des prix d'Allaire à *http://www.allaire.com*.[4]

Autres astuces finales

Si tous ces conseils ne vous ont pas aidé à trouver votre bogue, voici quelques astuces finales sur le débogage des servlets :

- Utilisez `System.getProperty("java.class.path")` dans votre servlet pour déboguer les problèmes de chemin de classes. Les servlets étant souvent exécutées à partir de serveurs web avec des JVM embarquées, il peut être difficile d'identifier exactement le chemin de classes utilisé par la JVM. La propriété `java.class.path` vous l'indiquera.

4. Nous ne serions pas très surpris si Allaire supprimait bientôt le support de ServletDebugger. Si cela arrivait, ou même si cela n'arrivait pas, surveillez la sortie d'une version en *open source*.

- Sachez que les classes qui se trouvent dans le chemin de classes direct du serveur (*racine_serveur/classes*) ne seront probablement pas rechargées, pas plus que, sur la plupart des serveurs, ne le seront les classes utilitaires dans le répertoire des classes de l'application web (*WEB-INF/classes*). Normalement, seules les classes de servlet sous le répertoire des classes de l'application web seront rechargées.
- Demandez au navigateur de montrer le contenu de la page qu'il affiche. Cela peut vous aider à identifier des problèmes de mise en forme. C'est généralement une option du menu Affichage.
- Assurez-vous que le navigateur ne mette pas dans un cache la sortie de la requête précédente en forçant un rechargement complet de la page. Dans Netscape Navigator, utilisez Shift-Reload ; dans Internet Explorer utilisez Shift-Refresh.
- Si vous écrasez la version de la méthode `init()` qui prend un `ServletConfig` en argument, vérifiez que la méthode surchargée commence par appeler `super.init(config)`.

Ajuster les performances

Le réglage des performances des servlets demande une réflexion quelque peu différente de celle demandée pour régler les performances des applications Java normales ou des applets. La raison est que la JVM exécutant les servlets est supposée gérer simultanément des dizaines voire des centaines de threads, chacun exécutant une servlet. Ces servlets coexistantes doivent partager les ressources de la JVM d'une manière différente des applications normales. Bien sûr, les astuces habituelles sont toujours valides, mais elles ont un impact différent lorsqu'elles sont utilisées avec un système hautement multithread. Voici donc des astuces qui ont un impact plus important sur les développeurs de servlets.

Avancer, mais sans prospérer

Évitez la création inutile d'objets. Cela a toujours été un bon conseil ; créer des objets inutilement consomme de la mémoire et pas mal de temps. Avec les servlets, c'est encore un meilleur conseil. Toutes les JVM, sauf les plus récentes, ont une mémoire de tas globale pour les objets qui doit être verrouillé pour chaque nouvelle allocation mémoire. Pendant qu'une servlet crée un nouvel objet ou alloue de la mémoire, aucune autre servlet ne peut faire de même.

Ne pas ajouter par concaténation

Évitez de concaténer plusieurs chaînes de caractères ensemble. Utilisez plutôt la méthode `append()` de `StringBuffer`. Cela aussi a toujours été un bon conseil, mais avec les servlets il est particulièrement tentant d'écrire un code comme le suivant pour préparer une chaîne avant sa sortie :

```
String output;
output += "<TITLE>";
output += "Bonjour, " + user;
output += "</TITLE>";
```

Bien que ce code paraisse beau et propre, il s'exécute globalement comme s'il avait été écrit de la façon suivante, avec un `StringBuffer` et un nouveau `String` créés à chaque ligne :

```
String output;
output = new StringBuffer().append("<TITLE>").toString();
output = new StringBuffer(output).append("Bonjour, ").toString();
output = new StringBuffer(output).append(user).toString();
output = new StringBuffer(output).append("</TITLE>").toString();
```

Lorsque l'efficacité est importante, écrivez le code afin qu'il ressemble au suivant dans lequel un seul `StringBuffer` et un seul `String` sont créés :

```
StringBuffer buf = new StringBuffer();
buf.append("<TITLE>");
buf.append("Bonjour, ").append(user);
buf.append("</TITLE);
output = buf.toString();
```

Notez qu'utiliser un tableau d'octets est encore plus efficace.

Limiter la synchronisation

Synchronisez quand c'est nécessaire, mais pas plus. Chaque bloc synchronisé dans une servlet diminue le temps de réponse de la servlet. La même instance de servlet pouvant gérer de multiples requêtes concurrentes, elle doit, bien sûr, faire attention afin de protéger sa classe et ses variables d'instance par des blocs synchronisés. Cependant, pendant tout le temps que le thread de la requête passe dans un bloc synchronisé de la servlet, aucun autre thread ne peut entrer dans ce bloc. En général, il vaut donc mieux faire en sorte que ces blocs soient le plus petit possible.

Vous devriez aussi examiner le pire résultat de la contention des threads. Si le pire des cas est supportable (comme dans le cas de l'exemple du compteur dans le chapitre 3, *Cycle de vie d'une servlet*), vous pouvez envisager la suppression de tous les blocs de synchronisation. Envisagez aussi l'utilisation de l'interface `SingleThreadModel`, avec laquelle le serveur gère une réserve d'instances de servlet pour garantir que chaque instance est utilisée par au plus un thread à la fois. Les servlets qui implémentent `SingleThreadModel` n'ont pas besoin d'accès synchronisé à leurs variables d'instance.

Enfin, souvenez-vous que `java.util.Vector` et `java.util.Hashtable` sont toujours synchronisés de façon interne, alors que leurs équivalents `java.util.ArrayList` et `java.util.HashMap`, introduits dans le JDK 1.2, ne le sont que sur demande. Si votre `Vector` ou votre `Hashtable` n'a pas besoin de la synchronisation et si vous utilisez le JDK 1.2, remplacez-les par `ArrayList` ou `HashMap`.

Utiliser des tampons pour les entrées/sorties

Utilisez des tampons pour les entrées/sorties, les fichiers de stockage, les flots chargés à partir d'une base de données et ainsi de suite. Cela améliore toujours les performances, mais l'amélioration peut être particulièrement importante avec les servlets. La raison en est que lire et écrire une unité à la fois peut ralentir le serveur entier du fait des changements de contexte fréquents que cela implique. Heureusement, en général vous

n'avez pas besoin d'un tampon lorsque vous écrivez dans un `PrintWriter` ou un `ServletOutputStream`, ou lorsque vous lisez à partir d'un `BufferedReader` ou `ServletInputStream`. La plupart des implémentations des serveurs utilisent déjà un tampon pour ces flots.

Essayer d'utiliser un OutputStream

Pour les pages web basées sur le codage de caractères Latin-1, il est techniquement possible d'employer un `PrintWriter` ou un `ServletOutputStream` pour écrire sur le client. L'usage d'un `PrintWriter` est l'approche recommandée car il supporte l'internationalisation, mais, avec certains serveurs, l'emploi d'un `ServletOutputStream` améliore considérablement les performances et `ServletOutputStream` propose une longue liste de méthodes `print()` et `println()` issues de l'API Servlet 1.0 dans laquelle l'option `PrintWriter` n'était pas disponible. Soyez simplement prudent. Beaucoup de serveurs fonctionnent à l'inverse et, pour eux, un `PrintWriter` donne de meilleures performances. À moins d'être certain de votre plate-forme de déploiement et d'avoir comparé les temps d'exécution, conservez un `PrintWriter`.

Utiliser un outil de profilage

Il existe de nombreux outils de profilage Java qui peuvent faciliter l'identification des goulets d'étranglement dans votre code. Après tout, la plupart des problèmes de performances dans du code Java côté serveur ne sont pas à mettre sur le compte du langage ou de la JVM mais sur un ensemble de goulets d'étranglement ; il convient donc de les localiser. Ces outils d'analyse fonctionnent en arrière-plan et observent la façon dont votre serveur web traitent les requêtes. Ils indiquent où le temps est passé et la mémoire allouée. Les deux outils les plus connus sont OptimizeIt! d'Intuitive Systems (*http://www.optimizeit.com*) et JProbe de Sitraka, précédemment KL Group (*http://www.sitraka.com/jprobe*). Plusieurs JVM acceptent aussi des options sur la ligne de commande (`-prof` pour le JDK 1.1 et `-Xrunhprof` pour le JDK 1.2) afin de générer des informations de profilage de base. Pour exécuter le serveur avec une certaine charge, il existe des outils comme Apache JMeter (*http://java.apache.org/jmeter*).

20

Quoi de neuf dans l'API Servlet 2.3 ?

Peu avant que se livre parte à l'impression, Sun Microsystems a publié le brouillon final (*Proposed Final Draft*) de la spécification de l'API Servlet 2.3.[1] Il ne s'agit pas de la spécification dans sa forme finale ; le Proposed Final Draft se trouve juste avant le Final Release formel et des détails techniques peuvent encore changer. Cependant, ces modifications ne doivent pas être significatives ; en fait les fournisseurs de serveurs ont déjà commencé à implémenter les nouvelles fonctionnalités. Dans ce chapitre, je vais détailler tout ce qui a changé entre l'API 2.2 et l'API 2.3. J'expliquerai aussi les raisons des changements et montrerai comment écrire des servlets qui utilisent les nouvelles fonctionnalités.[2]

Changements dans l'API Servlet 2.3

Dans l'API Servlet 2.3, le noyau des servlets n'a pas été beaucoup touché. Cela indique que les servlets ont atteint un haut niveau de maturité. La majorité du travail a consisté en l'ajout de nouvelles fonctionnalités hors du noyau, dont :

- les servlets requièrent maintenant le JDK 1.2 ou ultérieur ;
- un mécanisme de filtre a (enfin) été créé ;
- les événements du cycle de vie de l'application ont été ajoutés ;
- un nouveau support pour l'internationalisation a été ajouté ;

1. Bien que cette spécification ait été publiée par Sun, l'API Servlet 2.3 a été développée par plusieurs personnes et plusieurs sociétés réunis au sein du groupe d'experts JSR-053, conformément au Java Community Process (JCP) Version 2.0. Ce groupe d'experts était dirigé par Danny Coward de Sun Microsystems.

2. Ce contenu est apparu pour la première fois dans l'article « Servlet 2.3: New Features Exposed » de Jason Hunter, publié par JavaWorld (*http://www.javaworld.com*), © ITworld.com, Inc., janvier 2001. Réimprimé ici avec leur autorisation. (Voir aussi *http://www.javaworld.com/jw-01-2001/jw-0126-servletapi.html*).

- la technique d'expression des dépendances inter-JAR a été formalisée ;
- les règles de chargement des classes ont été clarifiées ;
- de nouveaux attributs d'erreur et de sécurité ont été ajoutés ;
- la classe `HttpUtils` a été désapprouvée ;
- de nouvelles méthodes utilitaires ont été ajoutées ;
- plusieurs comportements de DTD ont été étendus et clarifiés.

D'autres clarifications ont été apportées, mais elles concernent principalement les fournisseurs de serveurs et non les programmeurs de servlets (excepté que les programmeurs constateront une meilleure portabilité), c'est pourquoi je ne vais pas entrer dans ces détails.

Avant de commencer ma description, je tiens à signaler que ce chapitre est basé sur le brouillon de la Version 2.3. La plupart des fonctionnalités décrites ici ne fonctionneront pas encore sur tous les serveurs. Si vous voulez les tester, je vous recommande de télécharger l'implémentation de référence officielle, Apache Tomcat 4.0. Elle est en *open source* et vous pouvez la télécharger gratuitement. Au moment de l'écriture de ce livre, Tomcat 4.0 est en version bêta ; son support de l'API 2.3 s'améliore, mais est toujours incomplet. Lisez le fichier `NEW_SPECS.txt` venant avec Tomcat 4.0 pour connaître sont niveau de support de toutes les nouvelles fonctionnalités de la spécification.

Servlets dans J2SE et J2EE

Une des premières choses à noter à propos de l'API Servlet 2.3 est que les servlets dépendent maintenant de Java 2 Platform, Standard Edition 1.2 (aussi connue sous le nom J2SE 1.2 ou JDK 1.2). Ce petit changement, mais important, signifie que vous pouvez maintenant utiliser les fonctionnalités de J2SE 1.2 dans vos servlets et être assuré qu'elles fonctionneront sur tous les conteneurs de servlets. Précédemment, vous pouviez exploiter les fonctionnalités de J2SE 1.2, mais les serveurs n'étaient pas obligés de les supporter.

L'API Servlet 2.3 est susceptible de devenir une partie fondamentale de Java 2 Platform, Enterprise Edition 1.3 (J2EE 1.3). La version précédente, l'API Servlet 2.2, faisait partie de J2EE 1.2. La seule différence notable est l'ajout à la DTD de *web.xml* de quelques balises obscures du descripteur de déploiement ayant un rapport avec J2EE : `<resource-env-ref>` pour supporter les « objets administrés », comme ceux requis par JMS (*Java Messaging System*), `<res-ref-sharing-scope>` pour permettre un accès partagé ou exclusif à une référence de ressource et `<run-as>` pour spécifier l'identité de sécurité de l'appelant d'un EJB. La plupart des auteurs de servlets ne sont pas concernés par ces balises J2EE ; vous pouvez en obtenir une description complète dans la spécification de J2EE 1.3.

Filtres

La partie la plus importante de l'API 2.3 est l'ajout des *filtres*. Les filtres sont des objets qui peuvent transformer une requête ou modifier une réponse. Notez que ce ne sont pas des servlets ; ils ne créent pas une réponse. Ce sont des préprocesseurs de la requête avant qu'elle n'atteigne une servlet et/ou des postprocesseurs de la réponse quittant une servlet. En fait, les filtres sont la version mature de l'ancien concept de chaînage de ser-

vlets. Un filtre peut :

- intercepter une invocation de servlet avant qu'elle ne soit appelée ;
- examiner une requête avant l'appel d'une servlet ;
- modifier les en-têtes et les données de la requête en fournissant une version personnalisée de l'objet de requête qui enveloppe la requête réelle ;
- modifier les en-têtes et les données de réponse en fournisssant une version personnalisée de l'objet de réponse qui enveloppe la réponse réelle ;
- intercepter l'invocation d'une servlet après qu'elle a été appelée.

Un filtre peut être configuré de manière à agir sur une servlet ou un groupe de servlets ; cette servlet ou ce groupe peuvent être filtrés par zéro ou plusieurs filtres. Parmi les utilisations des filtres, on trouve l'authentification, la journalisation et l'audit, la conversion d'images, la compression de données, le chiffrement, le découpage en éléments, la génération d'événements d'accès à une ressource, l'emploi de XSLT pour transformer le contenu XML ou les chaînes de type MIME (tout comme le chaînage de servlets).

Un filtre implémente `javax.servlet.Filter` et définit trois méthodes :

`void setFilterConfig(FilterConfig config)`

Cette méthode fixe l'objet de configuration du filtre.

`FilterConfig getFilterConfig()`

Cette méthode retourne l'objet de configuration du filtre.

`void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)`

Cette méthode effectue le travail de filtrage.

Le serveur appelle `setFilterConfig()` une fois afin de préparer le filtre, puis il appelle `doFilter()` pour les différentes requêtes. L'interface `FilterConfig` offre des méthodes pour retrouver le nom du filtre, ses paramètres d'initialisation et le contexte de servlet actif. Le serveur peut aussi passer `null` à `setFilterConfig()` pour indiquer que le filtre ne doit plus effectuer son travail.

Dans la version finale de l'API 2.3, la méthode `getFilterConfig()` devrait être enlevée et la méthode `setFilterConfig(FilterConfig config)` devrait être remplacée par `init(FilterConfig)` et `destroy()`.

Chaque filtre reçoit la requête et la réponse courantes dans sa méthode `doFilter()`, ainsi qu'un `FilterChain` contenant les filtres qu'il reste à appliquer. Dans la méthode `doFilter()`, un filtre peut faire ce qu'il veut de la requête et de la réponse. (Par exemple, il peut regrouper les données en appelant leurs méthodes ou envelopper les objets pour leur donner un nouveau comportement, comme nous le verrons plus loin.) Le filtre appelle ensuite `chain.doFilter()` afin de transférer le contrôle au filtre suivant. Quand l'appel revient, un filtre peut, à la fin de sa propre méthode `doFilter()`, effectuer un travail supplémentaire sur la réponse ; par exemple, il peut journaliser des informations sur la réponse. Si le filtre veut arrêter le traitement de la requête et prendre le contrôle de la réponse, il peut très bien décider de ne pas appeler le filtre suivant.

Un filtre peut envelopper les objets de requête et/ou de réponse afin d'offrir un comportement personnalisé, modifiant l'implémentation de certains appels de méthode pour influencer les traitements suivants de la requête. Pour cela, l'API Servlet 2.3 fournit les nouvelles classes `HttpServletRequestWrapper` et `HttpServletResponseWrapper`. Elles ont une implémentation par défaut de toutes les méthodes de requête et de réponse et, par défaut, elles délèguent les appels à la requête et la réponse originales. Pour modifier le comportement d'une méthode, il suffit d'étendre l'enveloppe et de surcharger la méthode. Les enveloppes donnent au filtre un très bon contrôle du processus de traitement de la requête et de génération de la réponse. Voici le code d'un filtre de journalisation simple qui enregistre la durée d'exécution de chaque requête :

```
public class LogFilter implements Filter {

  FilterConfig config;

  public void setFilterConfig(FilterConfig config) {
    this.config = config;
  }

  public FilterConfig getFilterConfig() {
    return config;
  }

  public void doFilter(ServletRequest req,
                       ServletResponse res,
                       FilterChain chain) {
    ServletContext context = getFilterConfig().getServletContext();
    long bef = System.currentTimeMillis();
    chain.doFilter(req, res); // Aucun paramètre de chaînage nécessaire ici.
    long aft = System.currentTimeMillis();
    context.log("Requête à " + req.getRequestURI() + " : " + (aft-bef));
  }
}
```

Quand le serveur appelle `setFilterConfig()`, le filtre sauvegarde une référence à sa configuration dans la variable `config`, utilisée ensuite dans la méthode `doFilter()` pour retrouver le `ServletContext`. La logique de `doFilter()` est simple : calcul du temps de traitement de la requête et journalisation une fois le traitement achevé. Pour utiliser ce filtre, vous devez le déclarer dans le descripteur de déploiement *web.xml* à l'aide de la balise `<filter>` :

```
<filter>
  <filter-name>
    log
  </filter-name>
  <filter-class>
    LogFilter
  </filter-class>
</filter>
```

Cela indique au serveur qu'un filtre nommé `log` est implémenté dans la classe `LogFilter`. Vous pouvez appliquer un filtre enregistré à certains motifs d'URL ou à des

noms de servlets à l'aide de la balise `<filter-mapping>` :

```
<filter-mapping>
  <filter-name>log</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Cela configure le filtre afin qu'il opère sur toutes les requêtes du serveur (statiques ou dynamiques), exactement ce que nous voulons pour notre filtre de journalisation. Si vous vous connectez à une page simple, la sortie dans le journal pourrait ressembler à ceci :

```
Requête à /index.jsp : 10
```

Événements du cycle de vie

La deuxième modification importante de l'API Servlet 2.3 est l'ajout des événements du cycle de vie de l'application. Les objets auditeurs peuvent ainsi être avertis lorsque des contextes de servlet et des sessions sont initialisés et détruits, ainsi que de l'ajout et la suppression d'attributs d'un contexte ou d'une session.

Les événements du cycle de vie d'une servlet fonctionnent comme les événements de Swing. Tout auditeur intéressé par le cycle de vie de `ServletContext` peut implémenter l'interface `ServletContextListener`. Cette interface offre deux méthodes :

`void contextInitialized(ServletContextEvent e)`

Appelée quand une application web est prête pour la première fois à traiter les requêtes (c'est-à-dire au démarrage du serveur web et quand un contexte est ajouté ou rechargé). Les requêtes ne seront pas traitées tant que cette méthode n'est pas terminée.

`void contextDestroyed(ServletContextEvent e)`

Appelée quand une application web va s'arrêter (c'est-à-dire à l'arrêt du serveur web ou quand un contexte est supprimé ou rechargé). Le traitement des requêtes sera arrêté avant que cette méthode soit appelée.

La classe `ServletContextEvent` passée aux deux méthodes offre simplement une méthode `getServletContext()` qui retourne le contexte en cours d'initialisation ou de destruction.

Un auditeur intéressé par le cycle de vie des attributs de `ServletContext` peut implémenter l'interface `ServletContextAttributesListener` qui offre trois méthodes :

`void attributeAdded(ServletContextAttributeEvent e)`

Appelée quand un attribut est ajouté à un contexte de servlet.

`void attributeRemoved(ServletContextAttributeEvent e)`

Appelée quand un attribut est supprimé d'un contexte de servlet.

`void attributeReplaced(ServletContextAttributeEvent e)`

Appelée quand un attribut est remplacé par un autre attribut dans un contexte de servlet.

La classe `ServletContextAttributeEvent` étend `ServletContextEvent` et ajoute les méthodes `getName()` et `getValue()` afin que l'auditeur puisse être informé des modifications sur les attributs. C'est utile car les applications web ayant besoin de synchroniser leur état (attributs du contexte) avec des ressources telles que les bases de données peuvent maintenant le faire en un seul endroit.

Le modèle d'auditeur de session est similaire à celui d'auditeur de contexte. Dans la version session, il existe une interface `HttpSessionListener` ayant deux méthodes :

`void sessionCreated(HttpSessionEvent e)`

Appelée quand une session est créée.

`void sessionDestroyed(HttpSessionEvent e)`

Appelée quand une session est détruite (invalidée).

Les méthodes acceptent une instance de `HttpSessionEvent` avec une méthode d'accès `getSession()` pour retourner la session en cours de création ou de destruction. Vous pouvez utiliser toutes ces méthodes dans l'implémentation d'une interface d'administration afin de garder la trace des utilisateurs actifs dans l'application web.

Le modèle pour les sessions a aussi une interface `HttpSessionAttributesListener` avec trois méthodes. Ces méthodes signalent à l'auditeur quand des attributs sont modifiés et elles peuvent servir, par exemple, à une application qui synchronise les données de profil contenues dans les sessions avec une base de données :

`void attributeAdded(HttpSessionBindingEvent e)`

Appelée quand un attribut est ajouté à la session.

`void attributeRemoved(HttpSessionBindingEvent e)`

Appelée quand un attribut est supprimé de la session.

`void attributeReplaced(HttpSessionBindingEvent e)`

Appelée quand un attribut est remplacé par un autre attribut dans la session.

Comme vous devez vous y attendre, la classe `HttpSessionBindingEvent` étend `HttpSessionEvent` et ajoute les méthodes `getName()` et `getValue()`. La seule chose quelque peu anormale est que la classe d'événement s'appelle `HttpSessionBindingEvent`, au lieu de `HttpSessionAttributeEvent`. Les raisons en sont historiques ; l'API disposait déjà d'une classe `HttpSessionBindingEvent`, elle a donc été réutilisée. Ce point perturbant de l'API pourrait disparaître dans la version finale.

Une utilisation pratique des événements du cycle de vie pourrait être une connexion partagée à une base de données gérée par un auditeur du contexte. Vous pouvez déclarer l'auditeur dans la fichier *web.xml* de la façon suivante :

```
<listener>
 <listener-class>
   com.acme.MyConnectionManager
 </listener-class>
</listener>
```

Le serveur crée une instance de la classe de l'auditeur recevant les événements et utilise l'introspection pour détermine quelle(s) interface(s) d'auditeur la classe implémente. Gardez à l'esprit que puisque l'auditeur est configuré dans le descripteur de déploiement, vous pouvez ajouter de nouveaux auditeurs sans modifier le code. Voici com-

ment l'auditeur pourrait être écrit :

```
public class MyConnectionManager implements ServletContextListener {

  public void contextInitialized(ServletContextEvent e) {
    Connection con =     // Créer une connexion.
      e.getServletContext().setAttribute("con", con);
  }

  public void contextDestroyed(ServletContextEvent e) {
    Connection con =
       (Connection) e.getServletContext().getAttribute("con");
    try { con.close(); }
    catch (SQLException ignored) { } // Fermer la connexion.
  }
}
```

Cet auditeur s'assure qu'une connexion de base de données est disponible dans chaque nouveau contexte de servlet et que toutes les connexions sont fermées lorsque le contexte s'arrête.

L'interface `HttpSessionActivationListener`, une autre nouvelle interface d'auditeur dans l'API 2.3, est conçue de manière à gérer les sessions qui migrent d'un serveur à un autre. Un auditeur implémentant `HttpSessionActivationListener` est averti lorsqu'une session va être passivée (déplacée) et lorsque la session va être activée sur le deuxième hôte. Ces méthodes donnent à l'application la possibilité de rendre persistantes entre des JVM des données non sérialisables ou d'associer ou de dissocier des objets sérialisés à une sorte de modèle d'objets avant ou après la migration. L'interface a deux méthodes :

`void sessionWillPassivate(HttpSessionEvent e)`

La session va être passivée. La session sera toujours hors service quand cet appel est effectué.

`void sessionDidActivate(HttpSessionEvent e)`

La session va être activée. La session ne sera pas encore en service lorsque cet appel est effectué.

L'enregistrement de l'auditeur se fait de la même façon que pour les autres. Cependant, contrairement aux autres, les appels sur passivation et sur activation se produiront très certainement sur deux serveurs différents !

Sélectionner les codages de caractères

L'API Servlet 2.3 fournit des méthodes très utiles pour le support des langues étrangères dans l'envoi des formulaires. Une nouvelle méthode, `request.setCharacterEncoding(String encoding)`, permet d'indiquer au serveur le codage des caractères d'une requête. Un codage de caractères (aussi connu sous le nom de *charset*) est une manière de relier des octets à des caractères. Le serveur peut employer un charset pour analyser correctement les paramètres et les données POST.

Par défaut, un serveur analyse les paramètres à l'aide du charset Latin-1 (ISO 8859-1). Malheureusement, cela ne fonctionne que pour les langues occidentales. Quand un navigateur utilise un autre charset, il est supposé envoyer les informations de codage dans

l'en-tête Content-Type de la requête, mais pratiquement aucun navigateur ne s'y conforme. La méthode setCharacterEncoding() permet à une servlet d'indiquer au serveur quel charset est en cours d'utilisation (il s'agit en général du charset de la page qui contient le formulaire) ; le serveur s'occupe du reste. Par exemple, une servlet qui reçoit des paramètres en japonais à partir d'un formulaire codé en Shift_JIS peut lire ces paramètres de la manière suivante :

```
// Mettre le charset à Shift_JIS.
req.setCharacterEncoding("Shift_JIS");

// Lire un paramètre à l'aide de ce charset.
String name = req.getParameter("name");
```

N'oubliez pas de définir le codage avant d'appeler getParameter() ou getReader(). L'appel à setCharacterEncoding() peut lancer une java.io.UnsupportedEncodingException si le codage n'est pas supporté. Cette fonctionnalité est aussi disponible aux utilisateurs de l'API 2.2 et antérieure par le biais de la classe com.oreilly.servlet.ParameterParser.

Dépendances avec les fichiers JAR

Le plus souvent, un fichier WAR (fichier *Web Application Archive*, ajouté à l'API Servlet 2.2) a besoin que d'autres bibliothèques JAR existent sur le serveur et fonctionnent correctement. Par exemple, une application web utilisant la classe ParameterParser a besoin de cos.jar dans son chemin de classes. Une application web utilisant WebMacro a besoin de webmacro.jar.

Avant l'API 2.3, soit ces dépendances devaient être documentées (comme si quelqu'un lisait vraiment la documentation !), soit chaque application web devait inclure tous les fichiers JAR requis dans son propre répertoire *WEB-INF/lib* (grossissant inutilement chaque application web). L'API Servlet 2.3 permet d'exprimer des dépendances avec les fichiers JAR au sein du fichier WAR à l'aide de l'entrée *META-INF/MANIFEST.MF*. C'est la manière standard de déclarer des dépendances dans les fichiers JAR, mais, avec l'API 2.3, les fichiers WAR doivent officiellement supporter ce même mécanisme. Si une dépendance ne peut être satisfaite, un serveur peut poliment rejeter l'application web au moment du déploiement au lieu de générer un message d'erreur flou à l'exécution. Le mécanisme autorise un fort degré de granularité. Par exemple, il est possible d'exprimer une dépendance avec une version particulière d'un paquetage optionnel et le serveur se doit de trouver la bonne version avec un algorithme de recherche.

Chargeurs de classes

Voici une petite modification dont l'impact est important : dans l'API 2.3, un conteneur de servlets (le serveur) s'assure que les classes d'une application web ne voient les classes d'implémentation du serveur. Autrement dit, les chargeurs de classes doivent rester distincts.

Cela ne semble pas très important, mais les possibilités de collision entre les classes de l'application web et celles du serveur sont ainsi éliminées. Ce type de collisions est devenu un problème sérieux du fait des conflits entre les analyseurs syntaxiques XML. Chaque serveur a besoin d'un analyseur XML pour traiter les fichiers *web.xml* et plusieurs

applications web utilisent aussi aujourd'hui un analyseur XML pour la lecture, la manipulation et l'écriture de données XML. Si les analyseurs supportent des versions différentes de DOM ou SAX, les conflits peuvent être irréparables. La séparation des portées des classes résout très bien ce problème.

Nouveaux attributs d'erreur

La version précédente de l'API, l'API Servlet 2.2, a introduit plusieurs attributs de requête que les servlets et les pages JSP agissant comme cible d'une règle `<error-page>` peuvent utiliser. Souvenez-vous que nous avons dit précédemment dans ce livre que les règles `<error-page>` vous permettent de configurer une application web afin que certains codes d'erreur ou certains types d'exception provoque l'affichage de pages spécifiques :

```
<web-app>
    <!-- ..... -->
    <error-page>
        <error-code>
            404
        </error-code>
        <location>
            /404.html
        </location>
    </error-page>
    <error-page>
        <exception-type>
            javax.servlet.ServletException
        </exception-type>
        <location>
            /servlet/ErrorDisplay
        </location>
    </error-page>
    <!-- ..... -->
</web-app>
```

Une servlet se trouvant dans l'élément `<location>` d'une règle `<error-page>` peut recevoir les trois attributs suivants :

`javax.servlet.error.status_code`

Un `Integer` indiquant le code d'état de l'erreur, s'il existe.

`javax.servlet.error.exception_type`

Une instance de `Class` indiquant le type de l'exception qui a provoqué l'erreur, s'il existe.

`javax.servlet.error.message`

Un `String` donnant le message de l'exception, passé au constructeur de l'exception.

Avec ces attributs, une servlet peut générer une page d'erreur adaptée à l'erreur :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class ErrorDisplay extends HttpServlet {

  public void doGet(HttpServletRequest req,
      HttpServletResponse res)
      throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    String code = null, message = null, type = null;
    Object codeObj, messageObj, typeObj;

    // Retrouver les trois attributs d'erreur possibles. Certains
    // peuvent être null.

    codeObj = req.getAttribute("javax.servlet.error.status_code");
    messageObj = req.getAttribute("javax.servlet.error.message");
    typeObj = req.
        getAttribute("javax.servlet.error.exception_type");

    // Convertir les attributs en chaînes de caractères. Nous faisons
    // ainsi car certains anciens serveurs retournent des types String
    // alors que des serveurs plus récents retournent des types Integer,
    // String et Class. Ce code fonctionne sur tous les serveurs.

    if (codeObj != null) code = codeObj.toString();
    if (messageObj != null) message = messageObj.toString();
    if (typeObj != null) type = typeObj.toString();

    // La raison de l'erreur est, soit le code d'erreur, soit le type
    // d'exception.
    String reason = (code != null ? code : type);

    out.println("<HTML>");
    out.println("<HEAD><TITLE>" + reason + " : " + message +
        "</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H1>" + reason + "</H1>");
    out.println("<H2>" + message + "</H2>");
    out.println("<HR>");
    out.println("<I>Erreur lors de l'accès à " + req.getRequestURI() +
        "</I>");
    out.println("</BODY></HTML>");
  }
}
```

Mais que faire si une page d'erreur peut contenir la trace de la pile de l'exception ou l'URI de la servlet qui a provoqué le problème (puisque ce n'est pas toujours l'URI demandée à l'origine) ? Avec l'API 2.2, ce n'était pas possible. Avec l'API 2.3, ces informations sont disponibles dans deux nouveaux attributs :

`javax.servlet.error.exception`

Un objet `Throwable` qui est en fait l'exception lancée.

`javax.servlet.error.request_uri`

Un `String` donnant l'URI de la ressource ayant provoqué les problèmes.

Ces attributs permettent d'inclure dans la page d'erreur la trace de la pile de l'exception et l'URI de la ressource à problème. La servlet ci-dessous a été écrite de manière à utiliser ces nouveaux attributs. Notez qu'elle échoue proprement s'ils n'existent pas, pour des raisons de compatibilité :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ErrorDisplay extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    String code = null, message = null, type = null, uri = null;
    Object codeObj, messageObj, typeObj;
    Throwable throwable;

    // Retrouver les trois attributs d'erreur possibles. Certains
    // peuvent être null.
    codeObj = req.getAttribute("javax.servlet.error.status_code");
    messageObj = req.getAttribute("javax.servlet.error.message");
    typeObj = req.getAttribute("javax.servlet.error.exception_type");
    throwable = (Throwable)
      req.getAttribute("javax.servlet.error.exception");
    uri = (String)
      req.getAttribute("javax.servlet.error.request_uri");

    if (uri == null) {
      uri = req.getRequestURI(); // Dans ce cas, pas d'URI.
    }

    // Convertir les attributs en chaînes de caractères.
    if (codeObj != null) code = codeObj.toString();
    if (messageObj != null) message = messageObj.toString();
    if (typeObj != null) type = typeObj.toString();

    // La raison de l'erreur est, soit le code d'erreur, soit le type
    // d'exception.
    String reason = (code != null ? code : type);

    out.println("<HTML>");
    out.println("<HEAD><TITLE>" + reason + ": " + message +
                "</TITLE></HEAD>");
```

```
      out.println("<BODY>");
      out.println("<H1>" + reason + "</H1>");
      out.println("<H2>" + message + "</H2>");
      out.println("<PRE>");
      if (throwable != null) {
        throwable.printStackTrace(out);
      }
      out.println("</PRE>");
      out.println("<HR>");
      out.println("<I>Erreur d'accès à " + uri + "</I>");
      out.println("</BODY></HTML>");
    }
  }
```

Nouveaux attributs de sécurité

L'API Servlet 2.3 ajoute aussi deux nouveaux attributs de requête qui peuvent aider une servlet dans sa gestion des connexions HTTPS sûres. Pour les requêtes effectuées avec HTTPS, le serveur fournit les deux nouveaux attributs suivants :

`javax.servlet.request.cipher_suite`

Un `String` représentant la suite de chiffrement employée par HTTPS, si elle existe.

`javax.servlet.request.key_size`

Un `Integer` représentant la taille en bit de l'algorithme, si elle existe.

Un servlet peut se servir de ces attributs pour décider si la connexion est suffisamment sûre ou non. Une application peut refuser les connexions si le nombre de bits est trop faible ou si elle n'a pas confiance en certains algorithmes. Par exemple, une servlet peut utiliser la méthode suivante pour s'assurer que la connexion emploie une clé d'au moins 128 bits :

```
public boolean isAbove128(HttpServletRequest req) {
  Integer size = (Integer)
    req.getAttribute("javax.servlet.request.key_size");

  if (size == null || size.intValue() < 128) {
    return false;
  }
  else {
    return true;
  }
}
```

Le nom des attributs dans le brouillon final de la spécification utilise des tirets au lieu du caractère souligné ; cependant, ils sont modifiés comme montré ici dans la version finale afin d'être plus cohérents avec les noms d'attributs existants.

Petits ajustements

Un petit nombre de modifications ont aussi été apportées à la version 2.3 de l'API. Tout d'abord, la méthode `getAuthType()` qui retourne le type d'authentification utilisé pour l'identification d'un client a été redéfinie afin de retourner une des quatre nouvelles constantes de type `String` définies par la classe `HttpServletRequest` : `BASIC_AUTH`, `DIGEST_AUTH`, `CLIENT_CERT_AUTH` et `FORM_AUTH`. Cela permet de simplifier le code :

```
if (req.getAuthType() == req.BASIC_AUTH) {
  // Gérer l'authentification de base.
}
```

Bien entendu, les quatre constantes ont toujours des valeurs `String` traditionnelles. Le code suivant de l'API 2.2 fonctionne donc encore, mais il n'est pas aussi rapide ni aussi élégant. Notez le contrôle inversé avec `equals()` afin d'éviter une `NullPointerException` si `getAuthType()` retourne `null` :

```
if ("BASIC".equals(req.getAuthType())) {
  // Gérer l'authentification de base.
}
```

Une autre modification dans l'API 2.3 est la désapprobation de `HttpUtils`. `HttpUtils` a toujours été un ensemble étrange de méthodes statiques, des appels parfois utiles mais dont la place étaient ailleurs. Comme vous vous en souvenez peut-être, la classe contenait des méthodes permettant de reconstruire une URL originale à partir d'un objet de requête et de convertir les données des paramètres en une table de hachage. L'API 2.3 a déplacé cette fonctionnalité dans l'objet de requête et a désapprouvé `HttpUtils`. Voici les nouvelles méthodes de l'objet de requête :

`StringBuffer req.getRequestURL()`

Retourne un `StringBuffer` contenant l'URL de requête originale, reconstruite à partir des informations sur la requête.

`java.util.Map req.getParameterMap()`

Retourne un `Map` immuable contenant les paramètres de la requête. Les noms des paramètres servent de clés pour en retrouver les valeurs. Rien n'a été décidé quant aux paramètres ayant des valeurs multiples ; elles seront sans doute retournées dans un `String[]`. Ces méthodes utilisent la nouvelle méthode `req.setCharacterEncoding()` pour la conversion des caractères.

L'API 2.3 ajoute aussi deux nouvelles méthodes à `ServletContext` qui permettent d'obtenir le nom du contexte et une liste de ses ressources :

`String context.getServletContextName()`

Retourne le nom du contexte tel qu'il est déclaré dans le fichier *web.xml*.

`java.util.Set context.getResourcePaths()`

Retourne tous les chemins de ressource disponibles dans le contexte, sous forme d'un ensemble d'objets `String` immuables. Chaque `String` inclut une barre oblique de début (`/`) et doit être vue comme relative à la racine du contexte.

Il existe aussi une nouvelle méthode dans l'objet de réponse permettant d'augmenter le contrôle du programmeur sur le tampon de réponse. L'API 2.2 avait introduit la métho-

de `res.reset()` pour réinitialiser la réponse en effaçant son corps, ses en-têtes et son code d'état. L'API 2.3 ajoute la méthode `res.resetBuffer()` qui efface juste le corps de la réponse :

`void resetBuffer()`

Efface le tampon de réponse sans effacer les en-têtes ni le code d'état. Si la réponse a déjà été validée, une `IllegalStateException` est lancée.

Pour finir, après un long débat au sein du groupe d'experts, l'API Servlet 2.3 a été clarifiée à propos de ce qui se passe exactement lorsqu'une servlet s'exécutant dans un contexte non racine appelle `res.sendRedirect("/index.html")`. Le problème est que l'API Servlet 2.2 impose qu'un chemin incomplet tel que `"/index.html"` soit traduit par le conteneur de servlets en un chemin complet, mais rien ne dit comment les chemins de contexte doivent être gérés. Si la servlet effectuant l'appel est dans un contexte dont le chemin est `"/chemin_contexte"`, l'URI de redirection doit-il être converti relativement à la racine du conteneur (`http://serveur:port/index.html`) ou relativement à la racine du contexte (`http://serveur:port/chemin_contexte/index.html`) ? Pour conserver une portabilité maximale, il est impératif de définir le comportement ; après de longs débats, les experts ont choisi de faire la conversion relativement à la racine du conteneur. Pour ceux qui voudrait la racine du contexte, vous pouvez ajouter la sortie de `getContextPath()` à l'URI.

Clarifications de la DTD

L'API Servlet 2.3 lève quelques ambiguïtés concernant le comportement du descripteur de déploiement *web.xml*. Il est maintenant obligatoire de supprimer les espaces des valeurs textuelles dans le fichier *web.xml* avant utilisation. (Dans le XML standard non validé, tous les espaces sont généralement préservés.) Cette règle garantit que les deux entrées suivantes sont traitées de façon identique :

```
<servlet-name>bonjour<servlet-name>
```

et

```
<servlet-name>
  bonjour
</servlet-name>
```

L'API Servlet 2.3 autorise aussi une règle `<auth-constraint>` afin que la valeur spéciale `"*"` puisse être utilisée comme caractère générique de `<role-name>` afin d'accepter tous les rôles. Cela permet d'écrire une règle comme la suivante, qui laisse tous les utilisateurs entrer à partir du moment où ils ont été correctement identifiés comme appartenant à un rôle de l'application web :

```
<auth-constraint>
  <role-name>*</role-name> <!-- Accepte tous les rôles reconnus. -->
</auth-constraint>
```

Pour finir, il est maintenant clair qu'un nom de rôle déclaré par une règle `<security-role>` peut être employé comme paramètre de la méthode `isUserInRole()`. Par exemple, examinez l'extrait du fichier *web.xml* suivant :

```
<servlet>
    <servlet-name>
        secret
```

```
        </servlet-name>
        <servlet-class>
            SalaryViewer
        </servlet-class>
        <security-role-ref>
            <role-name>
                mgr <!-- Nom utilisé par la servlet. -->
            </role-name>
            <role-link>
                manager <!-- Nom utilisé par le descripteur de déploiement. -->
            </role-link>
        </security-role-ref>
    </servlet>

    <!-- ... -->

    <security-role>
        <role-name>
            manager
        </role-name>
    </security-role>
```

Ainsi, la servlet secrète peut appeler `isUserInRole("mgr")` ou `isUserInRole("manager")` ; les deux ont le même comportement. Globalement, `security-role-ref` agit comme créateur d'un alias, mais ce n'est pas nécessaire. C'est ce à quoi l'on pourrait logiquement s'attendre, mais la spécification de l'API 2.2 ne peut être lue de manière à impliquer que seuls les rôles déclarés explicitement dans une règle d'alias `<security-role-ref>` peuvent être utilisés. (Si vous ne comprenez pas cela, ce n'est pas important ; sachez simplement que les choses fonctionnent maintenant comme elles le doivent.)

Conclusion

L'API Servlet 2.3 inclut un nouveau mécanisme de filtrage très intéressant, un modèle de cycle de vie étendu et de nouvelles fonctionnalités pour le support de l'internationalisation, de la gestion des erreurs, des connexions sûres et des rôles d'utilisateurs. Le document de spécification a aussi été revu afin d'enlever les ambiguïtés qui pourraient interférer lors du déploiement sur des plates-formes différentes.

A

L'API Servlet, référence rapide

Le paquetage `javax.servlet` est le cœur de l'API Servlet. Il inclut l'interface `Servlet` de base que toutes les servlets doivent implémenter sous une forme ou sous une autre, et une classe abstraite `GenericServlet` pour développer des servlets basiques. Ce paquetage inclut aussi des classes pour la communication entre le serveur hôte et le client (`ServletRequest` et `ServletResponse`) et pour la communication avec le client (`ServletInputStream` et `ServletOutputStream`). La hiérarchie de classes du paquetage `javax.servlet` est donnée à la *figure A-1*. Les servlets doivent se limiter aux classes de ce paquetage quand le protocole sous-jacent est inconnu.

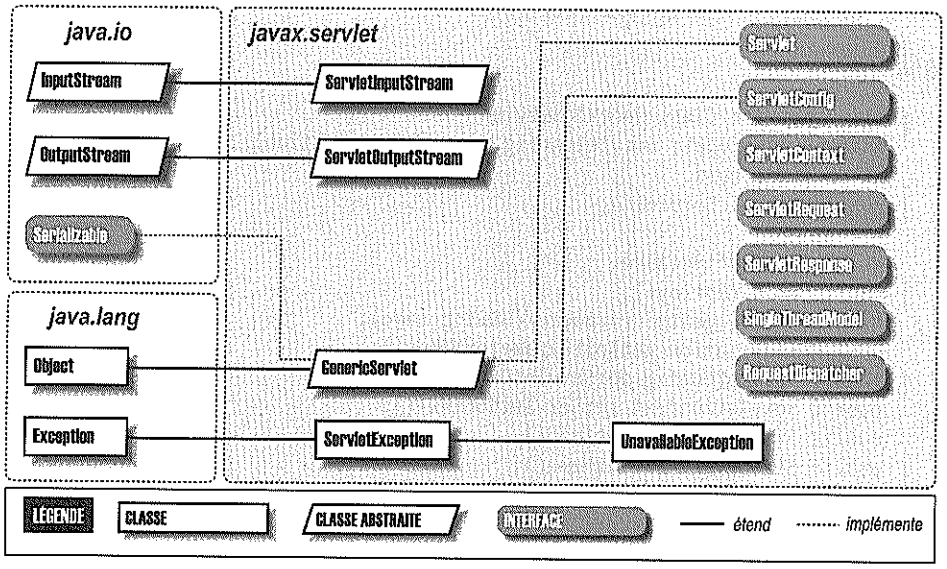


Figure A-1. Le paquetage java.servlet

GenericServlet

Synopsis

Nom de classe :	`javax.servlet.GenericServlet`
Classe mère :	`java.lang.Object`
Sous-classes directes :	`javax.servlet.http.HttpServlet`
Interfaces implémentées :	`javax.servlet.Servlet`, `javax.servlet.ServletConfig`, `java.io.Serializable`
Disponibilité :	API Servlet 1.0 et ultérieure

Description

`GenericServlet` fournit une implémentation de base de l'interface `Servlet` pour des servlets indépendantes du protocole. Pour vous aider, elle implémente aussi l'interface `ServletConfig`. La plupart des développeurs de servlets étendent cette classe ou `HttpServlet`, au lieu d'implémenter directement l'interface `Servlet`.

`GenericServlet` inclut des versions basiques des méthodes `init()` et `destroy()`, qui effectuent des tâches d'initialisation et de terminaison de base, telles que gérer l'objet `ServletConfig` du serveur. Il est de bon ton qu'une servlet surchargeant une de ces méthodes appelle la méthode correspondante de la classe mère. `GenericServlet` inclut aussi une méthode `log()` fournissant un accès aisé aux fonctions de journalisation de `ServletContext`.

La méthode `service()` est déclarée abstraite et doit être surchargée. Les servlets bien écrites surchargent aussi `getServletInfo()`.

Résumé

```
public abstract class GenericServlet
 implements Servlet, ServletConfig, java.io.Serializable {
  // Constructeurs.
  public GenericServlet();

  // Méthodes d'instance.
  public void destroy();
  public String getInitParameter(String name);
  public Enumeration getInitParameterNames();
  public ServletConfig getServletConfig();
  public ServletContext getServletContext();
  public String getServletInfo();
  public String getServletName();                          // Nouvelle en 2.2
  public void init() throws ServletException;              // Nouvelle en 2.1
  public void init(ServletConfig config) throws ServletException;
  public void log(String msg);
  public void log(String msg, Throwable t);                // Nouvelle en 2.1
  public abstract void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;
}
```

Constructeurs

`GenericServlet()`

```
public GenericServlet()
```

Description

Le constructeur par défaut de `GenericServlet` ne fait rien. Les tâches d'initialisation de la servlet ne doivent pas être faites dans le constructeur mais dans `init()`.

Méthodes d'instance

`destroy()`

```
public void destroy()
```

Description

Appelée par le conteneur de servlets pour indiquer à une servlet qu'elle a été mise hors service. Cette méthode est appelée après que tous les threads au sein de la méthode de service de la servlet sont terminés ou après une certaine période. Après l'appel de cette méthode par le conteneur de servlets, ce dernier n'appellera plus la méthode de service de la servlet. L'implémentation par défaut journalise la destruction de la servlet à l'aide de la méthode `log()`. Une servlet peut surcharger cette méthode afin de sauvegarder son état, libérer ses ressources (connexions à une base de données, threads, queues, etc.) et ainsi de suite.

`getInitParameter()`

```
public String getInitParameter(String name)
```

Description

Retourne la valeur du paramètre d'initialisation de la servlet indiqué ou `null` si aucun paramètre correspondant n'est trouvé. Vient de l'interface `ServletConfig`.

`getInitParameterNames()`

```
public Enumeration getInitParameterNames()
```

Description

Retourne les noms de tous les paramètres d'initialisation de la servlet sous forme d'un `Enumeration` d'objets `String` ou d'un `Enumeration` vide si la servlet n'a pas de paramètres. Vient de l'interface `ServletConfig`.

`getServletConfig()`

```
public ServletConfig getServletConfig()
```

Description

Retourne l'objet `ServletConfig` de la servlet. En pratique, cette méthode est rarement appelée par un `GenericServlet` car toutes les méthodes de `ServletConfig` sont dupliquées en interne.

`getServletContext()`

```
public ServletContext getServletContext()
```

Description
Retourne l'objet ServletContext de la servlet. Vient de l'interface ServletConfig.

getServletInfo()

```
public String getServletInfo()
```

Description
Retourne un String prédéfini par le programmeur décrivant la servlet. Une servlet doit surcharger cette méthode et fournir une chaîne d'identification personnalisée (par exemple, « Servlet panneau d'affichage de Al v1.21 »), mais ce n'est pas obligatoire.

getServletName()

```
public String getServletName()
```

Description
Retourne le nom de cette instance de servlet. Le nom peut être fourni *via* l'administration du serveur, peut être attribué dans le descripteur de déploiement de l'application web ou est le nom de classe pour une instance de la servlet non enregistrée (donc sans nom). Vient de l'interface ServletConfig. Cette méthode a été introduite dans l'API Servlet 2.2.

init()

```
public void init() throws ServletException;
public void init(ServletConfig config) throws ServletException
```

Description
Appelée par le serveur après que la servlet a été chargée pour la première fois et avant que sa méthode service() ne soit appelée. Une servlet peut surcharger cette méthode pour effectuer d'autres configurations, créer des ressources, etc. Une servlet écrite avec l'API Servlet 2.1 ou ultérieure peut implémenter la version sans argument. Les servlets qui doivent rester compatibles avec l'API Servlet 2.0 doivent implémenter la version prenant en argument un ServletConfig. N'implémentez pas les deux versions. L'implémentation par défaut d'init() journalise l'initialisation de la servlet et stocke l'objet ServletConfig afin qu'il puisse être utilisé par les méthodes de l'interface ServletConfig. Une servlet implémentant la version prenant en argument un ServletConfig doit appeler super.init(config) avant d'exécuter tout autre code d'initialisation. Les servlets exploitant la nouvelle version sans argument n'ont pas à le faire.

log()

```
public void log(String msg)
public void log(String msg, Throwable t)
```

Description
Écrit dans le journal de la servlet le message passé, précédé du nom de la servlet appelante. L'emplacement du journal est spécifique à chaque serveur, il s'agit généralement d'un fichier de journalisation des événements.

service()

```
public abstract void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException
```

Description

Appelée pour traiter une seule requête cliente. Une servlet reçoit les informations de la requête *via* un objet `ServletRequest` et retourne les données au client *via* un objet `ServletResponse`. C'est la seule méthode qui doit être surchargée lorsque l'on étend `GenericServlet`.

RequestDispatcher

Synopsis

Nom de l'interface :	`javax.servlet.RequestDispatcher`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 2.1 et ultérieure

Description

Interface d'un objet qui peut en interne distribuer les requêtes à toute ressource (comme une servlet, un fichier HTML ou un fichier JSP) sur le serveur. Le conteneur de servlets crée l'objet `RequestDispatcher` ensuite utilisé comme enveloppe d'une ressource serveur indiquée par un chemin particulier ou donnée par un nom particulier. Cette interface est conçue pour envelopper des servlets et des pages JSP, mais un conteneur de servlets peut créer des objets `RequestDispatcher` pour envelopper tout type de ressources. La distribution peut servir à renvoyer une requête à la ressource ou pour inclure le contenu de cette ressource à la réponse courante. Cette classe a été introduite dans l'API Servlet 2.1.

Déclaration

```
public interface RequestDispatcher {
  // Méthodes.
  public abstract void forward(ServletRequest req, ServletResponse res)
    throws ServletException, java.io.IOException        // Nouvelle en 2.1
  public abstract void include(ServletRequest req, ServletResponse res)
    throws ServletException, java.io.IOException        // Nouvelle en 2.1
}
```

Méthodes

forward()

```
public abstract void forward(ServletRequest req, ServletResponse res)
  throws ServletException, IOException, IllegalStateException
```

Description

Renvoie la requête d'une servlet vers une autre ressource du serveur. Cette méthode permet à une servlet d'effectuer un traitement préliminaire de la requête et de laisser une autre ressource générer la réponse. Pour un `RequestDispatcher` obtenu *via* `getRequestDispatcher()`, les éléments et les paramètres de l'objet `ServletRequest` sont ajustés afin de correspondre au chemin de la ressource cible. Cette méthode doit être appelée avant que la réponse soit validée ; si la réponse a déjà été validée, cette méthode lance une `IllegalStateException`. La sortie non validée présente dans le tampon de réponse est automatiquement effacée avant le renvoi. Les paramètres de requête et de réponse doivent être les mêmes objets que ceux passés à la méthode de service de la servlet appelante. Cette méthode a été introduite dans l'API Servlet 2.1.

`include()`

```
public abstract void include(ServletRequest req, ServletResponse res)
  throws ServletException, IOException
```

Description

Inclut le contenu d'une ressource dans la réponse courante. Les éléments et les paramètres de chemin de `ServletRequest` restent les mêmes que ceux de l'appelant ; si la ressource incluse a besoin d'accéder à ses propres éléments et paramètres de chemin, elle peut les retrouver en utilisant les attributs de requêtes définis par le serveur, `javax.servlet.include.request_uri`, `javax.servlet.include.context_path`, `javax.servlet.include.servlet_path`, `javax.servlet.include.path_info` et `javax.servlet.include.query_string`. La servlet incluse ne peut modifier le code d'état de la réponse, ni définir des en-têtes ; tout changement est ignoré. Les paramètres de requête et de réponse doivent être les mêmes objets que ceux passés à la méthode de service de la servlet appelante. Cette méthode a été introduite dans l'API Servlet 2.1.

Servlet

Synopsis

Nom de l'interface :	`javax.servlet.Servlet`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	`javax.servlet.GenericServlet`
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Toutes les servlets implémentent l'interface `Servlet`, soit directement, soit en étendant la classe `GenericServlet` ou `HttpServlet`. La plupart des développeurs de servlets trouvent qu'il est plus facile d'étendre l'une de ces deux classes que d'implémenter directement cette interface. L'interface déclare les fonctionnalités de base d'une servlet, l'initialisation, le traitement d'une requête cliente et la destruction.

Déclaration

```
public interface Servlet {
  // Méthodes.
  public abstract void destroy();
  public abstract ServletConfig getServletConfig();
  public abstract String getServletInfo();
  public abstract void init(ServletConfig config) throws ServletException;
  public abstract void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;
}
```

Méthodes

`destroy()`

```
public abstract void destroy()
```

Description

Appelée par le conteneur de servlets pour indiquer à une servlet qu'elle a été mise hors service. Voir la description complète dans `GenericServlet`.

`getServletConfig()`

```
public abstract ServletConfig getServletConfig()
```

Description

Retourne l'objet `ServletConfig` sauvegardé par la méthode `init()`.

`getServletInfo()`

```
public abstract String getServletInfo()
```

Description

Retourne un `String` défini par le programmeur décrivant la servlet.

`init()`

```
public abstract void init(ServletConfig config) throws ServletException
```

Description

Appelée par le conteneur de servlets après qu'elle a été chargée pour la première fois et avant que la méthode `service()` de la servlet ne soit appelée. Voir la description complète dans `GenericServlet`.

`service()`

```
public abstract void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException
```

Description

Appelée pour traiter une seule requête cliente. Une servlet reçoit les informations de la requête *via* l'objet `ServletRequest` et retourne les données au client *via* l'objet `ServletResponse`.

ServletConfig

Synopsis

Nom de l'interface :	`javax.servlet.ServletConfig`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	`javax.servlet.GenericServlet`
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Les conteneurs de servlets utilisent les objets `ServletConfig` pour passer aux servlets des informations d'initialisation et du contexte. L'information d'initialisation consiste généralement en une série de paramètres d'initialisation et un objet `ServletContext`, qui fournit les informations sur l'environnement du serveur. Une servlet peut implémenter `ServletConfig` afin d'offrir un accès aisé aux paramètres d'initialisation et à l'information de contexte, ce que fait `GenericServlet`.

Déclaration

```
public interface ServletConfig {
  // Méthodes.
  public abstract String getInitParameter(String name);
  public abstract Enumeration getInitParameterNames();
  public abstract ServletContext getServletContext();
  public abstract String getServletName();              // Nouvelle en 2.2
}
```

Méthodes

`getInitParameter()`

```
public abstract String getInitParameter(String name)
```

Description

Retourne la valeur du paramètre d'initialisation de la servlet indiqué ou `null` s'il n'existe pas.

`getInitParameterNames()`

```
public abstract Enumeration getInitParameterNames()
```

Description

Retourne le nom de tous les paramètres d'initialisation sous forme d'un `Enumeration` d'objets `String` ou un `Enumeration` vide s'il n'existe pas de paramètres.

`getServletContext()`

```
public abstract ServletContext getServletContext()
```

Description

Retourne l'objet `ServletContext` de cette servlet, permettant une interaction avec le conteneur de servlets.

getServletName()

```
public String getServletName()
```

Description

Retourne le nom de cette instance de servlet. Le nom peut être fourni *via* l'administration du serveur, peut être attribué dans le descripteur de déploiement de l'application web ou est le nom de classe pour une instance de la servlet non enregistrée (donc sans nom). Vient de l'interface ServletConfig. Cette méthode a été introduite dans l'API Servlet 2.2.

ServletContext

Synopsis

Nom de l'interface :	`javax.servlet.ServletContext`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

L'interface ServletContext définit un ensemble de méthodes que l'on peut utiliser pour communiquer avec le conteneur de servlets de manière non spécifique à une requête. Cela inclut trouver l'information de chemin, accéder aux autres servlets s'exécutant sur le serveur et écrire dans le journal du serveur. Chaque application web a un contexte de servlet différent.

Déclaration

```
public interface ServletContext {
  // Méthodes.
  public abstract Object getAttribute(String name);
  public abstract Enumeration getAttributeNames();       // Nouvelle en 2.1
  public abstract
    ServletContext getContext(String uripath);           // Nouvelle en 2.1
  public abstract String getInitParameter(String name); // Nouvelle en 2.2
  public abstract Enumeration getInitParameterNames();  // Nouvelle en 2.2
  public abstract int getMajorVersion();                 // Nouvelle en 2.1
  public abstract String getMimeType(String file);
  public abstract int getMinorVersion();                 // Nouvelle en 2.1
  public abstract
    RequestDispatcher getNamedDispatcher(String name);   // Nouvelle en 2.2
  public abstract String getRealPath(String path);
  public abstract URL getResource(String path)           // Nouvelle en 2.1
    throws MalformedURLException;
  public abstract
    InputStream getResourceAsStream(String path);        // Nouvelle en 2.1
```

```
    public abstract String getServerInfo();
    public abstract Servlet getServlet(String name)        // Désapprouvée
      throws ServletException;
    public abstract Enumeration getServletNames();         // Désapprouvée
    public abstract Enumeration getServlets();             // Désapprouvée
    public abstract
      void log(Exception exception, String msg);           // Désapprouvée
    public abstract void log(String msg);
    public abstract void log(String msg, Throwable t);     // Nouvelle en 2.1
    public abstract void removeAttribute(String name);     // Nouvelle en 2.1
    public abstract
       void setAttribute(String name, Object o);           // Nouvelle en 2.1
}
```

Méthodes

`getAttribute()`

```
public abstract Object getAttribute(String name)
```

Description

Retourne la valeur de l'attribut du contexte indiqué sous forme d'un `Object` ou `null` si l'attribut n'existe pas. Les attributs spécifiques au serveur peuvent être définis automatiquement par le conteneur de servlets afin de donner aux servlets des informations en plus de celle fournies par l'API Servlet de base. Les attributs peuvent aussi être fixés par programmation par les servlets afin de partager des informations au sein de l'application web représentée par ce contexte. Les noms d'attributs doivent suivre la même convention de nommage que les paquetages. Les noms `java.*` et `javax.*` sont réservés à la division Java Software de Sun Microsystems (précédemment JavaSoft) et `com.sun.*` est réservé par Sun Microsystems. Consultez la documentation de votre serveur pour la liste de ses attributs. Souvenez-vous que les servlets reposant sur des attributs spécifiques au serveur ne sont pas portables.

`getAttributeNames()`

```
public abstract Enumeration getAttributeNames()
```

Description

Retourne les noms de tous les attributs du contexte courant sous forme d'un `Enumeration` d'objets `String`. L'`Enumeration` retourné est vide si le contexte n'a pas d'attributs. Cette méthode a été introduite dans l'API Servlet 2.1.

`getContext()`

```
public abstract ServletContext getContext(String uripath)
```

Description

Retourne l'instance de `ServletContext` attribuée au chemin d'URI indiqué. Le chemin donné doit être absolu (commencer par `/`) et il est interprété relativement à la racine du serveur. Cette méthode permet à une servlet d'obtenir l'accès à un contexte différent du sien, la laissant voir les données au sein de ce contexte ou lui permettant d'obtenir un `RequestDispatcher` vers les ressources de ce con-

texte. Dans un environnement soucieux de la sécurité ou distribué, le conteneur de servlets peut retourner `null` pour tout chemin. Cette méthode a été introduite dans l'API Servlet 2.1.

`getInitParameter()`

```
public abstract String getInitParameter(String name)
```

Description

Retourne la valeur du paramètre d'initialisation du contexte indiqué ou `null` s'il n'existe pas. Cette méthode permet de rendre disponibles à une application web entière les informations de configuration utiles. Par exemple, elle peut fournir l'adresse e-mail du webmestre ou le nom d'un système qui détient des données critiques. Les paramètres d'initialisation du contexte sont donnés dans le descripteur de déploiement de l'application web. Cette méthode a été introduite dans l'API Servlet 2.2.

`getInitParameterNames()`

```
public abstract Enumeration getInitParameterNames()
```

Description

Retourne les noms de tous les paramètres d'initialisation du contexte sous forme d'un `Enumeration` d'objets `String`. Elle retourne un `Enumeration` vide si le contexte n'a pas d'attributs. Les paramètres d'initialisation du contexte sont donnés dans le descripteur de déploiement de l'application web. Cette méthode a été introduite dans l'API Servlet 2.2.

`getMajorVersion()`

```
public abstract int getMajorVersion()
```

Description

Retourne le numéro de version principal de l'API Servlet que ce conteneur supporte. Par exemple, un conteneur implémentant l'API 2.2 retourne 2. Cette méthode a été introduite dans l'API Servlet 2.1.

`getMimeType()`

```
public abstract String getMimeType(String file)
```

Description

Retourne le type MIME du fichier indiqué ou `null` s'il n'est pas connu. Certaines implémentations retournent `text/plain` si le fichier spécifié n'existe pas. Les types MIME communs sont `text/html`, `text/plain`, `image/gif` et `image/jpeg`.

`getMinorVersion()`

```
public abstract int getMinorVersion()
```

Description

Retourne le numéro de version secondaire de l'API Servlet que ce conteneur supporte. Par exemple, un conteneur implémentant l'API 2.1 retourne 1. Cette méthode a été introduite dans l'API Servlet 2.1.

`getNamedDispatcher()`

```
public abstract RequestDispatcher getNamedDispatcher(String name)
```

Description

Retourne un `RequestDispatcher` pour distribuer vers une ressource désignée par son nom, au lieu d'un chemin. Cela permet de distribuer vers des ressources qui ne sont pas forcément disponibles publiquement. Les servlets (et les pages JSP) peuvent obtenir un nom *via* l'administration du serveur ou le descripteur de déploiement de l'application web. Cette méthode retourne `null` si, pour une raison ou pour une autre, le contexte ne peut renvoyer un distributeur. Cette méthode a été introduite dans l'API Servlet 2.2.

`getRealPath()`

```
public abstract String getRealPath(String path)
```

Description

Retourne le chemin réel sur le système de fichiers de tout « chemin virtuel » indiqué ou `null` si la traduction ne peut être faite (par exemple lorsque le fichier est sur un système de fichiers distant ou s'il ne se trouve que dans une archive *.war*). Si le chemin donné est `/`, la méthode retourne la racine des documents pour le contexte. Si le chemin donné est le même que celui retourné par `getPathInfo()`, la méthode retourne le même chemin réel que retournerait `getPathTranslated()`. Il n'y a pas d'équivalent CGI.

`getResource()`

```
public abstract URL getResource(String path)
```

Description

Retourne une URL pour la ressource associée au chemin indiqué. Le chemin doit commencer par `/` et il est interprété relativement à la racine du contexte. La méthode peut renvoyer `null` si aucune ressource ne peut être associée au chemin. Elle permet à un conteneur de servlets de rendre disponible aux servlets n'importe quelle ressource. Les ressources peuvent se trouver sur un système de fichiers local ou distant, dans une base de données ou dans un fichier *.war*. Certains conteneurs peuvent aussi autoriser l'écriture sur l'objet URL à l'aide des méthodes de la classe `URL`. Cette méthode doit être utilisée quand les ressources ne sont pas toutes des fichiers locaux, comme dans un environnement distribué (où un conteneur de servlets peut se trouver sur une machine différente de celle de la ressource) ou quand le contenu provient d'un fichier *.war* (dont les fichiers ne peuvent être accédés directement). Le contenu de la ressource est retourné tel quel. Sachez donc que la requête sur une page *.jsp* renvoie le code source de la page. Utilisez à la place un `RequestDispatcher` pour inclure les résultats de l'exécution d'une page JSP. Le but de cette méthode est différent de celui de `Class.getResource()` qui recherche des ressources d'après le chargeur de classes. Cette méthode ne peut utiliser les chargeurs de classes. Elle a été introduite dans l'API Servlet 2.1.

`getResourceAsStream()`

```
public abstract InputStream getResourceAsStream(String path)
```

Description

Retourne un `InputStream` pour lire le contenu de la ressource associée au chemin indiqué. le chemin doit commencer par `/` et il est interprété relativement à la racine du contexte. La méthode peut renvoyer `null` si aucune ressource ne

peut être associée au chemin. L'emploi de cette méthode est souvent plus pratique que celui de `getResource()` ; cependant, les méta-informations sur la ressource, comme la taille du contenu et son type, disponibles avec `getResource()` sont perdues quand on utilise cette méthode. Elle a été introduite dans l'API Servlet 2.1.

`getServerInfo()`

```
public abstract String getServerInfo()
```

Description

Retourne le nom et la version du logiciel serveur, séparés par une barre oblique (/). La valeur est la même que la variable CGI `SERVER_SOFTWARE`.

`getServlet()`

```
public abstract Servlet getServlet(String name) throws ServletException
```

Description

Retourne `null` dans l'API Servlet 2.1 et ultérieure. Dans les versions précédentes, cette méthode retournait la servlet chargée correspondant au nom donné ou `null` si la servlet n'est pas trouvée. Cette méthode a été désapprouvée et elle retourne `null` à partir de l'API Servlet 2.1 car un accès direct à une autre instance de servlet ouvre trop d'opportunités d'erreurs. En effet, les servlets peuvent être détruites à tout moment par le conteneur de servlets. Aucun objet autre que le conteneur ne doit donc détenir une référence directe à une servlet. De plus, si le serveur supporte la répartition de charge, les servlets sont distribuées sur plusieurs serveurs et il est donc impossible de retourner une référence locale à une servlet. Les servlets doivent collaborer à l'aide des attributs partagés de `ServletContext`. Techniquement, implémenter cette méthode pour qu'elle retourne `null` n'a pas d'impact sur la compatibilité ascendante car le conteneur de servlets pouvait toujours retourner `null`.

`getServletNames()`

```
public abstract Enumeration getServletNames()
```

Description

Retourne un `Enumeration` vide dans l'API Servlet 2.1 et ultérieure. Dans les versions précédentes, cette méthode retournait un `Enumeration` des noms des objets `Servlet` chargés dans le contexte. Elle a été désapprouvée et elle retourne un `Enumeration` vide à partir de l'API Servlet 2.1 pour les mêmes raisons que `getServlet()` a été désapprouvée et implémentée de façon à retourner `null`. Cette méthode a été introduite dans l'API Servlet 2.0.

`getServlets()`

```
public abstract Enumeration getServlets() throws ServletException
```

Description

Retourne un `Enumeration` vide dans l'API Servlet 2.1 et ultérieure. Dans les versions précédentes, cette méthode retournait un `Enumeration` des objets `Servlet` chargés dans le contexte. Elle a été désapprouvée dans l'API Servlet 2.0 en faveur de `getServletNames()`. Elle a été implémentée de façon à retourner un `Enumeration` vide dans l'API Servlet 2.1 afin de se conformer au comportement de `getServletNames()`.

log()

```
public abstract void log(String msg)
```

Description

Écrit le message indiqué dans le journal de la servlet. L'emplacement de la sortie dépend du serveur, c'est en général un fichier journalisant les événements.

```
public abstract void log(String msg, Throwable t)
```

Description

Écrit le message indiqué et la trace de la pile du Throwable dans le journal de la servlet. L'emplacement de la sortie dépend du serveur, c'est en général un fichier journalisant les événements. Cette méthode a été introduite dans l'API Servlet 2.1.

```
public abstract void log(Exception exception, String msg)
```

Description

Écrit le message indiqué et la pile de l'Exception dans le journal de la servlet. L'emplacement de la sortie est propre au serveur. Notez le placement non standard du paramètre optionnel Exception en première place au lieu de la dernière. Cette méthode a été désapprouvée dans l'API Servlet 2.1 en faveur de log(String msg, Throwable t), une méthode qui suit l'ordre standard des paramètres et accepte la journalisation de tout Throwable, pas uniquement un Exception. Cette méthode a été introduite dans l'API Servlet 2.0.

removeAttribute()

```
public abstract void removeAttribute(String name)
```

Description

Supprime du contexte l'attribut de nom indiqué. Les attributs doivent être supprimés lorsqu'ils ne sont plus utiles afin d'économiser la mémoire. Cette méthode a été introduite dans l'API Servlet 2.1.

setAttribute()

```
public abstract void setAttribute(String name, Object o)
```

Description

Associe, dans le contexte, le nom indiqué à l'objet. Toute association existante de même nom est remplacée. Cette méthode a été introduite dans l'API Servlet 2.1.

ServletException

Synopsis

Nom de classe :	javax.servlet.ServletException
Classe mère :	java.lang.Exception
Sous-classes directes :	javax.servlet.UnavailableException
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Une exception générique lancée par les servlets rencontrant des difficultés.

Résumé

```
public class ServletException extends java.lang.Exception {
  // Constructeurs.
  public ServletException();                                  // Nouvelle en 2.0
  public ServletException(String msg);
  public
    ServletException(String msg, Throwable rootCause);       // Nouvelle en 2.1
  public ServletException(Throwable rootCause);              // Nouvelle en 2.1
  public Throwable getRootCause();                           // Nouvelle en 2.1
}
```

Constructeurs

public ServletException()

```
public ServletException()
public ServletException(String msg)
public ServletException(String msg, Throwable rootCause)
public ServletException(Throwable rootCause)
```

Description

Construit une nouvelle `ServletException`, avec une explication et une « cause origine » de l'exception optionnelles. Si le message explicatif est donné, la méthode `getMessage()` permet de le retrouver ; si une cause origine est donnée, elle peut être retrouvée en appelant `getRootCause()`. Le message et la cause origine sont généralement inclus dans les journaux de la servlet et les messages d'erreur de l'utilisateur. Les versions du constructeurs prenant une cause origine en argument ont été introduits dans l'API Servlet 2.1.

Méthodes d'instance

getRootCause()

```
public Throwable getRootCause()
```

Description

Retourne l'objet `Throwable` qui a provoqué l'exception ou `null` s'il n'y a pas de cause origine. Cette méthode a été introduite dans l'API Servlet 2.1.

ServletInputStream

Synopsis

Nom de classe :	`javax.servlet.ServletInputStream`
Classe mère :	`java.io.InputStream`
Sous-classes directes :	aucune
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Fournit un flot d'entrée pour la lecture de données binaires à partir d'une requête cliente, y compris la méthode `readLine()` pour la lecture des données ligne par ligne. La méthode `getInputStream()` de `ServletRequest` retourne un `ServletInputStream`. Pour les servlets HTTP, le `ServletInputStream` donne accès aux données POST envoyées.

Résumé

```
public abstract class ServletInputStream extends java.io.InputStream {
  // Constructeurs.
  protected ServletInputStream();

  // Méthodes d'instance.
  public int readLine(byte b[], int off, int len) throws IOException;
}
```

Constructeurs

`ServletInputStream()`

```
protected ServletInputStream()
```

Description

Le constructeur par défaut ne fait rien. Une servlet ne doit jamais construire son propre `ServletInputStream`.

Méthodes d'instance

`readLine()`

```
public int readLine(byte b[], int off, int len) throws IOException
```

Description

Lit les octets du flot d'entrée dans le tableau d'octets `b`, en commençant à l'indice donné par `off`. Elle s'arrête de lire après un `\n` ou après avoir lu `len` octets. Le caractère de fin `\n` est aussi lu dans le tampon. Elle retourne le nombre d'octets lus ou -1 si la fin du flot a été atteinte. La version de cette classe fournie par l'API Servlet 2.0 a un bogue qui fait que le paramètre `len` est ignoré. La lecture d'une ligne plus longue que le tampon provoque donc une `ArrayIndexOutOfBoundsException`. Ce bogue a été corrigé dans l'API Servlet 2.1.

ServletOutputStream

Synopsis

Nom de classe :	`javax.servlet.ServletOutputStream`
Classe mère :	`java.io.OutputStream`
Sous-classes directes :	aucune
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Fournit un flot de sortie pour retourner des données binaires au client. Une servlet obtient un objet `ServletOutputStream` par la méthode `getOutputStream()` de `ServletResponse`. Bien qu'elle inclut des méthodes `print()` et `println()` pour envoyer du texte ou du HTML, `ServletOutputStream` a été remplacée par `PrintWriter`. Elle ne devrait être utilisée que pour envoyer des données binaires. Si vous étendez `ServletOutputStream`, vous devez fournir une implémentation de la méthode `write(int)`.

Résumé

```
public abstract class ServletOutputStream extends java.io.OutputStream {
  // Constructeurs.
  protected ServletOutputStream();

  // Méthodes d'instance.
  public void print(boolean b) throws IOException;
  public void print(char c) throws IOException;
  public void print(double d) throws IOException;
  public void print(float f) throws IOException;
  public void print(int i) throws IOException;
  public void print(long l) throws IOException;
  public void print(String s) throws IOException;
  public void println() throws IOException;
  public void println(boolean b) throws IOException;
  public void println(char c) throws IOException;
  public void println(double d) throws IOException;
  public void println(float f) throws IOException;
  public void println(int i) throws IOException;
  public void println(long l) throws IOException;
  public void println(String s) throws IOException;
}
```

Constructeurs

`ServletOutputStream()`

```
protected ServletOutputStream()
```

Description

Le constructeur par défaut ne fait rien.

Méthodes d'instance

`print()`

```
public void print(boolean b) throws IOException
public void print(char c) throws IOException
public void print(double d) throws IOException
public void print(float f) throws IOException
public void print(int i) throws IOException
public void print(long l) throws IOException
public void print(String s) throws IOException
```

Description

Écrit les données passées sur le client, sans inclure de retour chariot/saut de ligne (CRLF).

`println()`

```
public void println() throws IOException
public void println(boolean b) throws IOException
public void println(char c) throws IOException
public void println(double d) throws IOException
public void println(float f) throws IOException
public void println(int i) throws IOException
public void println(long l) throws IOException
public void println(String s) throws IOException
```

Description

Écrit les données sur le client, ajoutant un CRLF. La méthode sans paramètre écrit simplement un CRLF.

ServletRequest

Synopsis

Nom de l'interface :	`javax.servlet.ServletRequest`
Interface mère :	aucune
Sous-interfaces directes :	`javax.servlet.http.HttpServletRequest`
Implémentée par :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Un objet `ServletRequest` encapsule des informations à propos d'une seule requête cliente, dont les paramètres de la requête, les attributs de la requête, les locales du client et un flot d'entrée pour la lecture des données binaires à partir du corps de la requête. `ServletRequest` peut être étendue pour fournir des informations supplémentaires spécifiques au protocole. Par exemple, `HttpServletRequest` inclut des méthodes pour manipuler les en-têtes HTTP.

Déclaration

```
public interface ServletRequest {
  // Méthodes.
  public abstract Object getAttribute(String name);
  public abstract Enumeration getAttributeNames();       // Nouvelle en 2.1
  public abstract String getCharacterEncoding();         // Nouvelle en 2.0
  public abstract int getContentLength();
  public abstract String getContentType();
  public abstract ServletInputStream getInputStream() throws IOException;
  public abstract Locale getLocale();                    // Nouvelle en 2.2
  public abstract Enumeration getLocales();              // Nouvelle en 2.2
  public abstract String getParameter(String name);
  public abstract Enumeration getParameterNames();
  public abstract String[] getParameterValues(String name);
  public abstract String getProtocol();
  public abstract
    BufferedReader getReader() throws IOException;       // Nouvelle en 2.0
  public abstract String getRealPath(String path);       // Désapprouvée
  public abstract String getRemoteAddr();
  public abstract String getRemoteHost();
  public abstract
    RequestDispatcher getRequestDispatcher(String path);// Nouvelle
  public abstract String getScheme();
  public abstract String getServerName();
  public abstract int getServerPort();
  public abstract boolean isSecure();                    // Nouvelle en 2.2
  public abstract void removeAttribute(String name);     // Nouvelle en 2.2
  public abstract
    void setAttribute(String name, Object o);            // Nouvelle en 2.1
}
```

Méthodes

`getAttribute()`

```
public abstract Object getAttribute(String name)
```

Description

Retourne la valeur de l'attribut nommé sous forme d'`Object` ou `null` si l'attribut n'existe pas. Des attributs de requêtes spécifiques au serveur peuvent être définis automatiquement par le conteneur de servlets afin de fournir aux servlets des informations autres que celles fournies par l'API Servlet de base. Les attributs peuvent aussi être définis par le code des servlets afin de passer des informations d'une servlet à une autre par le biais d'un `RequestDispatcher`. Les attributs doivent suivre les mêmes conventions que les noms de paquetage. Les noms de paquetage `java.*` et `javax.*` sont réservés à la division Java Software de Sun Microsystems (précédemment JavaSoft) et `com.sun.*` est réservé à Sun Microsystems. Consultez la documentation de votre serveur pour obtenir la liste des attributs intégrés. Souvenez-vous que les servlets se reposant sur des attributs spécifiques à un serveur ne sont pas portables.

getAttributeNames()

```
public abstract Enumeration getAttributeNames()
```

Description

Retourne le nom de tous les attributs courants de la requête sous forme d'un Enumeration d'objets String. Elle retourne un Enumeration vide si la requête n'a pas d'attributs. Cette méthode a été introduite dans l'API Servlet 2.1.

getCharacterEncoding()

```
public abstract String getCharacterEncoding()
```

Description

Retourne le codage du jeu de caractères pour le flot d'entrée de la servlet ou null s'il n'est pas connu. Cette méthode a été introduite dans l'API Servlet 2.0.

getContentLength()

```
public abstract int getContentLength()
```

Description

Retourne la longueur, en octets, du contenu en train d'être envoyé *via* le flot d'entrée ou -1 si la longueur n'est pas connue (comme lorsqu'il n'y a pas de données). Équivalente à la variable CGI CONTENT_LENGTH.

getContentType()

```
public abstract String getContentType()
```

Description

Retourne le type du contenu en train d'être envoyé *via* le flot d'entrée ou null s'il n'est pas connu ou s'il n'y a pas de données. Identique à la variable CGI CONTENT_TYPE.

getInputStream()

```
public abstract ServletInputStream getInputStream()
  throws IOException, IllegalStateException
```

Description

Récupère le flot d'entrée sous forme d'objet ServletInputStream. ServletInputStream est une sous-classe directe de InputStream et elle peut être traitée de façon identique à un InputStream normal, avec en plus la possibilité de lire efficacement l'entrée ligne par ligne dans un tableau d'octets. Cette méthode doit être utilisée pour lire une entrée binaire. Elle lance une IllegalStateException si getReader() a déjà été appelée sur la requête. Il n'est pas nécessaire d'intercepter explicitement IllegalStateException.

getLocale()

```
public abstract Locale getLocale()
```

Description

Retourne la Locale préférée du client, prise dans l'en-tête de requête Accept-Language. Si la requête du client ne contient pas d'en-tête Accept-Language, cette méthode retourne la locale par défaut du serveur. Utilisez la classe com.oreilly.servlet.LocaleNegotiator pour avoir une détection plus fiable de la locale. Cette méthode a été introduite dans l'API Servlet 2.2.

getLocales()

```
public abstract Enumeration getLocales()
```

Description

Retourne un Enumeration des objets Locale indiquant les locales acceptées par le client comme indiqué par l'en-tête Accept-Language, dans l'ordre de préférence. Si la requête du client ne contient pas d'en-tête Accept-Language, cette méthode retourne un Enumeration contenant un Locale, la locale par défaut du serveur. Utilisez la classe com.oreilly.servlet.LocaleNegotiator pour avoir une détection plus fiable de la locale. Cette méthode a été introduite dans l'API Servlet 2.2.

getParameter()

```
public abstract String getParameter(String name)
```

Description

Retourne la valeur du paramètre nommé sous forme d'un String. Retourne null si le paramètre n'existe pas ou une chaîne vide si le paramètre existe mais qu'il n'a pas de valeur. La valeur est donnée dans sa forme normale décodée. Si le paramètre a plusieurs valeurs, utilisez la méthode getParameterValues() pour obtenir un tableau de valeurs. Si cette méthode est appelée sur un paramètre ayant plusieurs valeurs, la valeur retournée est la même que celle du premier élément du tableau renvoyé par getParameterValues(). Si l'information du paramètre vient avec des données POST codées, elle peut ne pas être disponible si les données POST ont déjà été lues manuellement en utilisant les méthodes getReader() ou getInputStream(). Cette méthode a été momentanément désapprouvée en faveur de getParameterValues(), mais grâce à une forte pression de la communauté des développeurs, elle a été remise dans l'API Servlet 2.0.

getParameterNames()

```
public abstract Enumeration getParameterNames()
```

Description

Retourne le nom des paramètres sous forme d'un Enumeration d'objets String. L'Enumeration est vide si la servlet n'a pas de paramètres.

getParameterValues()

```
public abstract String[] getParameterValues(String name)
```

Description

Retourne toutes les valeurs du paramètre nommé sous forme d'un tableau d'objets String ou null si le paramètre n'existe pas. Une valeur unique est retournée dans un tableau de taille 1.

getProtocol()

```
public abstract String getProtocol()
```

Description

Retourne le nom et la version du protocole utilisé par la requête dans un String de la forme *protocole/version-principale.version-secondaire*. Équivalent à la variable CGI SERVER_PROTOCOL.

getReader()

```
public abstract BufferedReader getReader()
  throws IOException, IllegalStateException
```

Description

Cette méthode retrouve le flot d'entrée sous forme d'un objet BufferedReader, qui doit être utilisé pour lire des données de type caractère puisque le lecteur traduit le jeu de caractères de façon adéquate. Cette méthode lance une IllegalStateException si getInputStream() a déjà été appelée sur la même requête. Elle lance une UnsupportedEncodingException si le codage des caractères de l'entrée n'est pas supporté ou inconnu. Cette méthode a été introduite dans l'API Servlet 2.0.

getRealPath()

```
public abstract String getRealPath(String path)
```

Description

Retourne le chemin réel sur le système de fichiers de tout « chemin virtuel » ou null la traduction ne peut se faire. Si le chemin donné est /, elle retourne la racine des documents du serveur. Si le chemin donné est le même que celui retourné par getPathInfo(), elle retourne le même chemin réel que retournerait getPathTranslated(). Il n'y a pas d'équivalent CGI. Cette méthode a été désapprouvée dans l'API Servlet 2.1 en faveur de la méthode getRealPath() de ServletContext.

getRemoteAddr()

```
public abstract String getRemoteAddr()
```

Description

Retourne l'adresse IP de la machine du client sous forme d'un String. Cette information vient de la socket connectant le serveur et le client, l'adresse distante peut donc être celle d'un proxy. Identique à la variable CGI REMOTE_ADDR.

getRemoteHost()

```
public abstract String getRemoteHost()
```

Description

Retourne le nom de la machine cliente. Cette information vient de la socket connectant le serveur et le client et peut donc être le nom d'un proxy. Identique à la variable CGI REMOTE_HOST.

getRequestDispatcher()

```
public abstract RequestDispatcher getRequestDispatcher(String path)
```

Description

Retourne un RequestDispatcher capable de distribuer la requête vers le chemin indiqué. Un RequestDispatcher peut servir à renvoyer une requête vers la ressource ou à inclure le contenu de cette ressource dans la réponse courante. La ressource peut être dynamique ou statique. Le nom de chemin indiqué peut être relatif, mais il ne peut sortir du contexte de servlet courant. Si le chemin commence par /, il est interprété relativement à la racine du contexte courant. Cette

méthode retourne null si le conteneur de servlets ne peut renvoyer un RequestDispatcher. À la différence de la méthode getRequestDispatcher() de ServletContext, cette méthode peut prendre un chemin relatif. Elle a été introduite dans l'API Servlet 2.2.

getScheme()

```
public abstract String getScheme()
```

Description

Cette méthode retourne le schéma utilisé pour effectuer la requête. Par exemple, http, https ou ftp, ou encore les nouveaux modes spécifiques à Java jdbc et rmi.

getServerName()

```
public abstract String getServerName()
```

Description

Retourne le nom du serveur qui a reçu la requête. C'est un attribut de ServletRequest car il peut changer pour chaque requête différente selon la manière dont le client fait référence au serveur. Équivalent à la variable CGI SERVER_NAME.

getServerPort()

```
public abstract int getServerPort()
```

Description

Retourne le numéro du port sur lequel cette requête a été reçue. Identique à la variable CGI SERVER_PORT.

isSecure()

```
public abstract boolean isSecure()
```

Description

Indique si cette requête a été faite à l'aide d'un canal sécurisé, comme HTTPS. Cette méthode a été introduite dans l'API Servlet 2.2.

removeAttribute()

```
public abstract void removeAttribute(String name)
```

Description

Supprime de la requête l'attribut de nom indiqué. En général, les attributs spécifiques au serveur ne peuvent être supprimés. Cette méthode a été introduite dans l'API Servlet 2.2.

setAttribute()

```
public abstract void setAttribute(String name, Object o)
```

Description

Associe dans la requête un objet au nom indiqué. Toute association de même nom existante est remplacée. Les attributs de requêtes servent le plus souvent à passer des informations d'une servlet à une autre lorsque l'on utilise un RequestDispatcher. Les noms d'attributs doivent suivre les mêmes conventions

que les noms de paquetage. Les noms de paquetage `java.*` et `javax.*` sont réservés à la division Java Software de Sun Microsystems (précédemment JavaSoft) et `com.sun.*` est réservé à Sun Microsystems. Cette méthode a été introduite dans l'API Servlet 2.1.

ServletResponse

Synopsis

Nom de l'interface :	`javax.servlet.ServletResponse`
Interface mère :	aucune
Sous-interfaces directes :	`javax.servlet.http.HttpServletResponse`
Implémentée par :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Les servlets utilisent des objets `ServletResponse` pour retourner des données MIME au client. Pour envoyer des données binaires, utilisez le `ServletOutputStream` retourné par `getOutputStream()`. Pour envoyer des données caractères, utilisez le `PrintWriter` retourné par `getWriter()`. Vous pouvez spécifier explicitement le type MIME de la sortie en utilisant la méthode `setContentType()` et définir la locale de la réponse à l'aide de la méthode `setLocale()`. Effectuez ces appels avant d'appeler `getWriter()` car `getWriter()` consulte le type de contenu et la locale pour déterminer le charset à utiliser. Consultez la RFC 2045 à *http://www.ietf.org/rfc/rfc2045.txt* pour plus d'informations sur MIME.

Déclaration

```
public interface ServletResponse {
  // Méthodes.
  public abstract void flushBuffer() throws IOException; // Nouvelle en 2.2
  public abstract int getBufferSize();                   // Nouvelle en 2.2
  public abstract String getCharacterEncoding();         // Nouvelle en 2.0
  public abstract Locale getLocale();                    // Nouvelle en 2.2
  public abstract ServletOutputStream getOutputStream()
    throws IOException, IllegalStateException;
  public abstract PrintWriter getWriter()                // Nouvelle en 2.0
    throws IOException, IllegalStateException;
  public abstract boolean isCommitted();                 // Nouvelle en 2.2
  public abstract
    void reset() throws IllegalStateException;           // Nouvelle en 2.2
  public abstract void setBufferSize(int size)           // Nouvelle en 2.2
    throws IllegalStateException;
  public abstract void setContentLength(int len);
  public abstract void setContentType(String type);
  public abstract void setLocale(Locale loc);            // Nouvelle en 2.2
}
```

Méthodes

`flushBuffer()`

```
public abstract void flushBuffer() throws IOException
```

Description

Force l'écriture sur le client du contenu du tampon. L'appel de cette méthode valide automatiquement la réponse. Le code d'état et les en-têtes sont donc écrits et un appel à reset() n'est plus possible. Cette méthode a été introduite dans l'API Servlet 2.2.

`getBufferSize()`

```
public abstract int getBufferSize()
```

Description

Retourne un `int` indiquant la taille courante du tampon ou `0` dans le cas improbable où la mise en tampon n'est pas utilisée. Cette méthode a été introduite dans l'API Servlet 2.2.

`getCharacterEncoding()`

```
public abstract String getCharacterEncoding()
```

Description

Retourne le code du charset utilisé pour ce corps MIME. Il s'agit du charset donné par une modification du type de contenu ou `ISO-8859-1` si aucun charset n'a été spécifié. Cette méthode a été introduite dans l'API Servlet 2.0.

`getLocale()`

```
public abstract Locale getLocale()
```

Description

Retourne le `Locale` actuellement affecté à la réponse. Cette méthode a été introduite dans l'API Servlet 2.2.

`getOutputStream()`

```
public abstract ServletOutputStream getOutputStream()
  throws IOException, IllegalStateException
```

Description

Retourne un `ServletOutputStream` pour écrire des données binaires (octet par octet). Aucun codage n'est effectué. Lance une `IllegalStateException` si `getWriter()` a déjà été appelée sur cette réponse.

`getWriter()`

```
public abstract PrintWriter getWriter() throws IOException
```

Description

Retourne un PrintWriter pour écrire des données caractères. L'écrivain code les caractères conformément au charset donné par le type du contenu. Si aucun charset n'est spécifié dans le type de contenu, comme c'est généralement le cas, l'écrivain utilise le codage ISO-8859-1 (Latin-1) approprié aux langues occidentales. Cette méthode lance une `IllegalStateException` si `getOutputStream()` a déjà été appelée sur cette réponse et une `UnsupportedEncodingException` si le

codage du flot de sortie n'est pas supporté ou inconnu. Cette méthode a été introduite dans l'API Servlet 2.0.

isCommitted()

```
public abstract boolean isCommitted()
```

Description

Retourne un boolean indiquant si une partie de la réponse a déjà été envoyée. Si cette méthode retourne true, il est trop tard pour changer le code d'état et les en-têtes. Cette méthode a été introduite dans l'API Servlet 2.2.

reset()

```
public abstract void reset() throws IllegalStateException
```

Description

Efface le tampon de réponse ainsi que le code d'état et les en-têtes de réponse courants. Cette méthode doit être appelée avant que la réponse ne soit validée, sinon elle lance une IllegalStateException. reset() est appelée automatiquement par les méthodes sendError() et sendRedirect(). Cette méthode a été introduite dans l'API Servlet 2.2.

setBufferSize()

```
public abstract void setBufferSize(int size) throws IllegalStateException
```

Description

Indique au serveur la taille minimale du tampon de réponse, en octets, que les servlets acceptent. Le serveur peut fournir un tampon plus grand, pour, par exemple, utiliser de blocs de 8 Ko afin de faciliter sa réutilisation. Un tampon plus important permet d'écrire plus de contenu avant qu'il ne soit envoyé au client ; la servlet dispose ainsi de plus de temps pour affecter les codes d'état et les en-têtes. Un tampon plus petit réduit la consommation mémoire et permet au client de recevoir les données plus tôt. Cette méthode doit être appelée avant que le corps de la réponse ne soit écrit, sinon elle lance une IllegalStateException. Cette méthode a été introduite dans l'API Servlet 2.2

setContentLength()

```
public abstract void setContentLength(int len)
```

Description

Définit la longueur du contenu retourné par le serveur. Dans les servlets HTTP, elle définit l'en-tête HTTP Content-Length. Les servlets HTTP utilisent cette méthode pour autoriser les connexions persistantes et renseigner l'indicateur de progression du client. Son utilisation est optionnelle. Si le contenu de la réponse tient entièrement dans le tampon de réponse, le serveur peut définir automatiquement la longueur du contenu.

setContentType()

```
public abstract void setContentType(String type)
```

Description

Cette méthode définit le type du contenu de la réponse au type spécifié. Dans les servlets HTTP, elle définit l'en-tête HTTP Content-Type.

setLocale()

```
public abstract void setLocale(Locale loc)
```

Description

Définit la locale de la réponse. Le serveur modifie les en-têtes `Content-Language` et `Content-Type` conformément à la locale indiquée. Cette méthode doit être appelée après avoir fixé `Content-Type` et avant d'appeler `getWriter()`. Par défaut, la locale de la réponse est la locale par défaut du serveur. Cette méthode a été introduite dans l'API Servlet 2.2.

SingleThreadModel

Synopsis

Nom de l'interface :	`javax.servlet.ServletConfig`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 2.0 et ultérieure

Description

`SingleThreadModel` est une interface de marquage sans méthodes. Si une servlet implémente cette interface, le serveur assure que chaque instance de la servlet ne traite qu'une requête à la fois. Par exemple, les serveurs peuvent implémenter cette fonctionnalité en maintenant une réserve d'instances de servlets et en distribuant les requêtes entrantes vers des servlets libres. L'emploi de `SingleThreadModel` rend la servlet sûre vis-à-vis des threads ; cependant, cette interface n'empêche en rien les problèmes de synchronisation pouvant survenir lorsque des servlets accèdent à des ressources partagées comme des variables de classe statiques ou des variables non locales à la servlet. Il existe quelques rares cas pour lesquels cette interface est utile.

Déclaration

```
public interface SingleThreadModel {
}
```

UnavailableException

Synopsis

Nom de classe :	`javax.servlet.UnavailableException`
Classe mère :	`javax.servlet.ServletException`
Sous-classes directes :	aucune
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Une servlet peut lancer une UnavailableException à tout moment pour indiquer qu'elle ne peut servir les requêtes du client. Il existe deux types d'indisponibilité : permanente (dans laquelle le problème ne peut se corriger de lui-même et où des actions administratives doivent être effectuées) et temporaire (dans laquelle le problème a de grandes chances de se corriger de lui-même après un certain temps). Pour marquer une servlet comme indisponible temporairement, indiquez une durée (en secondes) lorsque vous construisez l'exception. Les serveurs bien écrits utilisent cette information pour donner au client de meilleurs messages d'erreur. Les implémentations des servlets ne sont pas obligées de traiter différemment les indisponibilités temporaire et permanente.

Résumé

```
public class UnavailableException extends ServletException {
  // Constructeurs.
  public UnavailableException(int seconds, Servlet servlet,
    String msg);                                          // Désapprouvée
  public UnavailableException(Servlet servlet,
    String msg);                                          // Désapprouvée
  public UnavailableException(String msg);                // Nouvelle en 2.2
  public UnavailableException(String msg, int seconds); // Nouvelle en 2.2

  // Méthodes d'instance.
  public Servlet getServlet();                            // Désapprouvée
  public int getUnavailableSeconds();
  public boolean isPermanent();
}
```

Constructeurs

UnavailableException()

```
public UnavailableException(String msg)
public UnavailableException(String msg, int seconds)
public UnavailableException(Servlet servlet, String msg)
public UnavailableException(int seconds, Servlet servlet, String msg)
```

Description

Construit une UnavailableException avec un message explicatif. Vous pouvez aussi indiquer une période d'indisponibilité, donnée en secondes. Les deux constructeurs prenant un paramètre un Servlet ont été désapprouvés dans l'API Servlet 2.2 et remplacés par les versions plus simples et plus sûres qui n'acceptent pas d'instance d'une servlet.

Méthodes d'instance

getServlet()

```
public Servlet getServlet()
```

Description

Retourne la servlet qui a lancé cette exception ou `null` si l'instance de servlet n'a pas été fournie au constructeur. Cette méthode a été désapprouvée à partir de l'API Servlet 2.2 pour des raisons de sécurité.

`getUnavailableSeconds()`

```
public int getUnavailableSeconds()
```

Description

Retourne le nombre de secondes pendant lesquelles la servlet sera indisponible. Une valeur négative indique une indisponibilité permanente. Rien n'est fait pour essayer de compenser le temps passé depuis le lancement de l'exception.

`isPermanent()`

```
public boolean isPermanent()
```

Description

Retourne `true` si la servlet est indisponible indéfiniment, sinon `false`.

B

L'API Servlet HTTP, référence rapide

Le paquetage `javax.servlet.http` apporte le support des servlets conversant avec le protocole HTTP. Les classes de ce paquetage sont construites au-dessus des fonctionnalités de base fournies par le paquetage `javax.servlet` afin de fournir aux servlets un accès aux fonctionnalités propres à HTTP, comme les codes d'état, les en-têtes de requête et de réponse, les sessions et les cookies. La *figure B-1* montre la hiérarchie de classes du paquetage `javax.servlet.http`.

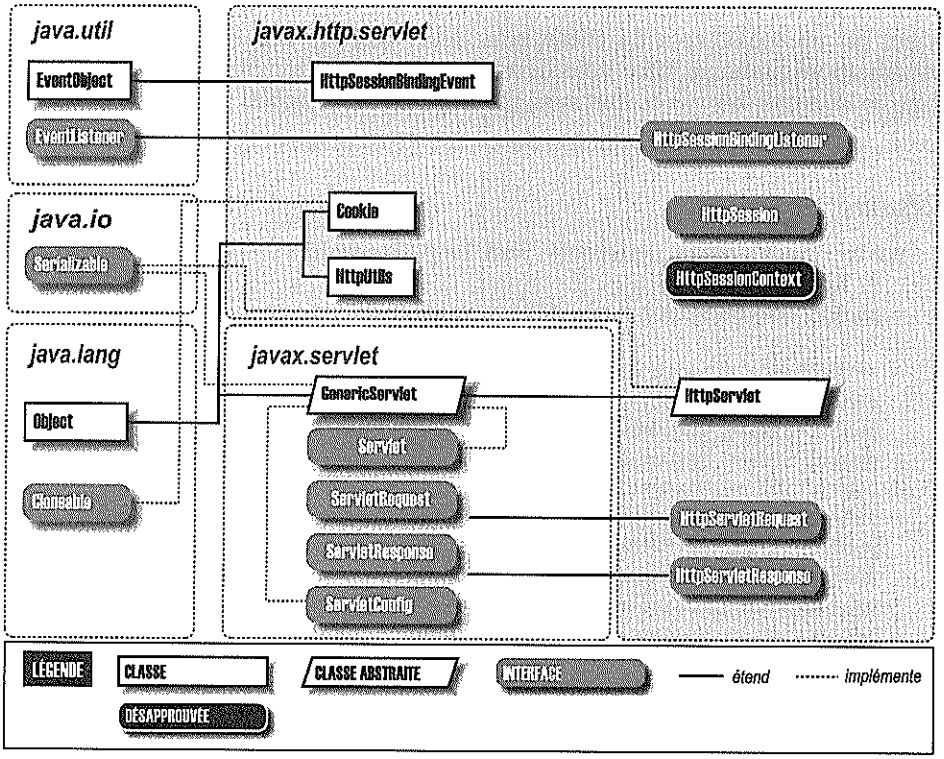


Figure B-1. Le paquetage javax.servlet.http

Cookie

Synopsis

Nom de classe :	`javax.servlet.http.Cookie`
Classe mère :	`java.lang.Object`
Sous-classes directes :	aucune
Interfaces implémentées :	`java.lang.Cloneable`
Disponibilité :	API Servlet 2.0 et ultérieure

Description

La classe `Cookie` fournit aux servlets un moyen facile de lire, créer et manipuler des cookies HTTP, qui permettent aux servlets de stocker une petite quantité de données sur le client. Les cookies sont en général utilisés pour le suivi de session ou pour stocker de petites quantités d'information de configuration spécifique à l'utilisateur. Pour plus d'informations, consulter le chapitre 7, *Suivi de session*.

Une servlet utilise la méthode `getCookies()` de `HttpServletRequest` pour retrouver les cookies soumis dans une requête cliente. La méthode `addCookie()` de `HttpServletResponse` envoie un nouveau cookie au navigateur. Les cookies étant spécifiés à l'aide les en-têtes HTTP, il faut appeler `addCookie()` avant que la réponse ne soit validée.

Les méthodes `getXXX()` sont rarement utilisées car quand un cookie est envoyé au serveur, il contient uniquement son nom, sa valeur et sa version. Si vous définissez un attribut sur un cookie reçu du client, vous devez l'ajouter à la réponse pour que les modifications prennent effet. Vous devez aussi savoir que tous les attributs, excepté le nom, la valeur et la version, sont aussi réinitialisés sur le cookie.

Cette classe supporte la spécification des cookie de Netscape et de la RFC 2109.

Résumé

```
public class Cookie implements java.lang.Cloneable {
  // Constructeurs.
  public Cookie(String name, String value);

  // Méthodes d'instance.
  public Object clone();
  public String getComment();
  public String getDomain();
  public int getMaxAge();
  public String getName();
  public String getPath();
  public boolean getSecure();
  public String getValue();
  public int getVersion();
  public void setComment(String purpose);
  public void setDomain(String pattern);
  public void setMaxAge(int expiry);
  public void setPath(String uri);
```

```
    public void setSecure(boolean flag);
    public void setValue(String newValue);
    public void setVersion(int v);
}
```

Constructeurs

Cookie()

```
public Cookie(String name, String value)
```

Description

Construit un nouveau cookie avec un nom et une valeur de départ. Les règles pour les noms et les valeurs sont données dans la spécification des cookies de Netscape et la RFC 2109.

Méthodes d'instance

clone()

```
public Object clone()
```

Description

Surcharge la méthode `clone()` standard pour retourner une copie de cet objet (un cookie dupliqué).

getComment()

```
public String getComment()
```

Description

Retourne le commentaire associé au cookie. Cette information est disponible uniquement après que le commentaire a été défini ; quand le cookie est renvoyé par le client, cette information n'est pas incluse.

getDomain()

```
public String getDomain()
```

Description

Retourne la limitation de domaine associée au cookie. Cette information est disponible uniquement après que le domaine a été défini ; quand le cookie est renvoyé par le client, cette information n'est pas incluse.

getMaxAge()

```
public int getMaxAge()
```

Description

Retourne l'âge maximal autorisé pour ce cookie. Cette information est disponible uniquement après que l'âge maximal a été défini ; quand le cookie est renvoyé par le client, cette information n'est pas incluse.

getName()

```
public String getName()
```

Description

Retourne le nom de ce cookie.

getPath()

```
public String getPath()
```

Description

Retourne la limitation de chemin pour cette servlet. Cette information est disponible uniquement après que le chemin a été défini ; quand le cookie est renvoyé par le client, cette information n'est pas incluse.

getSecure()

```
public boolean getSecure()
```

Description

Retourne `true` si ce cookie demande une connexion sécurisée, sinon `false`. Cette information est disponible uniquement après que l'indicateur a été défini ; quand le cookie est renvoyé par le client, cette information n'est pas incluse.

getValue()

```
public String getValue()
```

Description

Retourne la valeur de ce cookie, dans une chaîne de caractères

getVersion()

```
public int getVersion()
```

Description

Retourne la version de ce cookie.

setComment()

```
public void setComment(String purpose)
```

Description

Définit le champ de commentaire du cookie. Un commentaire décrit le rôle d'un cookie. Un navigateur web peut choisir d'afficher ce texte à l'utilisateur. Les commentaires ne sont pas supportés dans la version 0 des cookies.

setDomain()

```
public void setDomain(String pattern)
```

Description

Spécifie un motif de restriction de domaine. Un motif de domaine spécifie les serveurs qui verront les cookies. Par défaut, les cookies ne sont retournés qu'à la machine qui les a sauvés. Spécifier un motif de nom de domaine contourne cela. Le motif doit commencer par un point et doit en contenir au moins deux. Un motif ne correspond qu'à une entrée après le point initial. Par exemple, le motif `.foo.com` est valide et correspond à `www.foo.com` et `upload.foo.com` mais pas à `www.upload.foo.com`. Pour plus de détails sur les motifs de domaine, reportez-vous à la spécification des cookies de Netscape et à la RFC 2109.

`setMaxAge()`

```
public void setMaxAge(int expiry)
```

Description

Spécifie l'âge maximum du cookie en secondes avant qu'il n'expire. Une valeur négative indique le comportement par défaut, le cookie expirera lorsque le navigateur se terminera. Une valeur zéro indique au navigateur de supprimer le cookie immédiatement.

`setPath()`

```
public void setPath(String uri)
```

Description

Spécifie un chemin pour le cookie, qui est un sous-ensemble des URI à qui le cookie doit être envoyé. Par défaut, les cookies sont envoyés à la page qui définit le cookie et à toutes les pages dans ou sous ce répertoire. Par exemple, si */servlet/CookieMonster* définit un cookie, le chemin par défaut est `/servlet`. Ce chemin indique que le cookie doit être envoyé à */servlet/Elmo* et à */servlet/subdir/BigBird*, mais pas à l'alias de servlet */Oscar.html* ou à un programme CGI sous */cgi-bin*. Un chemin défini à « / » provoque l'envoi du cookie à toutes les pages du serveur. Un chemin de cookie doit être tel qu'il inclut la servlet qui a défini le cookie.

`setSecure()`

```
public void setSecure(boolean flag)
```

Description

L'indicateur de sécurité signale si le cookie ne doit être envoyé que *via* un canal sûr, comme SSL. Sa valeur est par défaut à `false`.

`setValue()`

```
public void setValue(String newValue)
```

Description

Donne une nouvelle valeur au cookie. Avec les cookies version 0, les valeurs ne doivent pas contenir les caractères suivants : espaces, crochets, parenthèses, signes égal, virgules, guillemets, barres obliques, points d'interrogation, signes @, points-virgule et deux-point. Les valeurs vides peuvent ne pas avoir le même comportement suivant les navigateurs.

`setVersion()`

```
public void setVersion(int v)
```

Description

Les servlets peuvent envoyer et recevoir des cookies conformes aux cookies persistants de Netscape (Version 0) ou aux cookies plus récents, encore expérimentaux, de la RFC 2109 (Version 1). Les cookies nouvellement créés sont par défaut des cookies version 0 afin de maximiser l'interopérabilité. Quand on utilise des cookies version 1, le conteneur de servlets peut aussi envoyer des cookies de type 0 ayant le même nom et la même valeur pour des raisons de compatibilité ascendante.

HttpServlet

Synopsis

Nom de classe :	`javax.servlet.http.HttpServlet`
Classe mère :	`javax.servlet.GenericServlet`
Sous-classes directes :	aucune
Interfaces implémentées :	`javax.servlet.Servlet`, `java.io.Serializable`
Disponibilité :	API Servlet 1.0 et ultérieure

Description

`HttpServlet` est une classe abstraite qui sert de classe de base aux servlets HTTP (World Wide Web). La méthode publique `service()` distribue les requêtes à une méthode `service()` protégée spécifique à HTTP, qui distribue ensuite les requêtes vers des fonctions de traitements particulières pour chaque type de soumission HTTP : `doGet()`, `doPost()` et ainsi de suite. Puisque l'implémentation par défaut des servlets HTTP prend en charge la distribution vers ces méthodes, si vous surchargez la méthode protégée `service()`, vous pouvez, soit traiter cette distribution manuellement, soit ne pas utiliser les fonctions de traitement pour les méthodes de requête HTTP.

Résumé

```
public abstract class HttpServlet extends javax.servlet.GenericServlet
  implements javax.servlet.Servlet, java.io.Serializable {
  // Constructeurs.
  public HttpServlet();

  // Méthode d'instance publique.
  public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;

  // Protected instance methods
  protected void doDelete(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;              // Nouvelle en 2.0
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;
  protected void doOptions(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;              // Nouvelle en 2.0
  protected void doPost(HttpServletRequest req,  HttpServletResponse res)
    throws ServletException, IOException;
  protected void doPut(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;              // Nouvelle en 2.0
  protected void doTrace(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;              // Nouvelle en 2.0
  protected long getLastModified(HttpServletRequest req);
  protected void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException;
}
```

Constructeurs

`HttpServlet()`

```
public HttpServlet()
```

Description

Le constructeur par défaut ne fait rien. Puisque vous ne pouvez pas être sûr du moment et de la façon dont les classes sont chargées, il n'est pas judicieux de surcharger ce constructeur pour effectuer des tâches d'initialisation. Utilisez plutôt `init()`.

Méthodes d'instance publiques

`service()`

```
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException
```

Description

Cette méthode `service()` gère la distribution des requêtes vers la méthode service() protégée spécifique à HTTP ; en général, elle ne doit pas être surchargée.

Méthodes d'instance protégées

`doDelete()`

```
protected void doDelete(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP DELETE à cette méthode. Les servlets implémentent cette méthode pour traiter les requêtes DELETE. L'implémentation par défaut retourne une erreur HTTP `SC_BAD_REQUEST`. Cette méthode a été introduite dans l'API Servlet 2.0.

`doGet()`

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP GET à cette méthode. Les servlets implémentent cette méthode pour traiter les requêtes GET. L'implémentation par défaut retourne une erreur HTTP `SC_BAD_REQUEST`.

`doOptions()`

```
protected void doOptions(HttpServletRequest req,
                         HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP OPTIONS à cette méthode. L'implémentation par défaut détermi-

ne quelles options sont supportées et retourne un en-tête approprié. Par exemple, si une servlet surcharge `doGet()` et `doPost()`, le navigateur est informé que GET, POST, HEAD, TRACE et OPTIONS sont supportés. Il n'y a presque jamais de raison de surcharger cette méthode. Cette méthode a été introduite dans l'API Servlet 2.0.

doPost()

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP POST à cette méthode. Les servlets implémentent cette méthode pour traiter les requêtes POST. L'implémentation par défaut retourne une erreur HTTP `SC_BAD_REQUEST`.

doPut()

```
protected void doPut(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP PUT à cette méthode. Les servlets implémentent cette méthode pour traiter les requêtes PUT. L'implémentation par défaut retourne une erreur HTTP `SC_BAD_REQUEST`. Reportez-vous à la RFC 2068 à *http://www.ietf.org/rfc/rfc2068.txt* pour plus d'informations sur les requêtes HTTP PUT. Cette méthode a été introduite dans l'API Servlet 2.0.

doTrace()

```
protected void doTrace(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

L'implémentation par défaut de `service()` dans `HttpServlet` distribue les requêtes HTTP TRACE à cette méthode. L'implémentation par défaut retourne un message donnant la liste de tous les en-têtes envoyés dans la requête TRACE. Il n'y a presque jamais de raison de surcharger cette méthode. Cette méthode a été introduite dans l'API Servlet 2.0.

getLastModified()

```
protected long getLastModified(HttpServletRequest req)
```

Description

Retourne la date et l'heure (exprimées en millisecondes depuis le 1er janvier 1970 GMT à minuit) auxquelles le contenu produit par la servlet a été modifié pour la dernière fois. Des valeurs négatives indiquent que le moment n'est pas connu. L'implémentation par défaut retourne -1. Appelée par les serveurs pour le support des requêtes HTTP GET conditionnelles et la gestion du cache des réponses. Pour plus d'informations, lisez le chapitre 4, *Retrouver l'information*.

service()

```
protected void service(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
```

Description

La méthode service() publique distribue les requêtes à cette méthode service(). Cette méthode gère la distribution des requêtes vers doGet(), doPost() et les autres fonctions de traitement en fonction du type de la requête. Si cette méthode est surchargée, aucune fonction de traitement n'est appelée.

HttpServletRequest

Synopsis

Nom de l'interface :	javax.servlet.http.HttpServletRequest
Super-interface :	javax.servlet.ServletRequest
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

HttpServletRequest étend la classe de base ServletRequest, fournissant des fonctionnalités supplémentaires pour les servlets HTTP (World Wide Web). Elle inclut le support des cookies et du suivi de session ainsi que l'accès aux informations d'en-têtes HTTP. HttpServletRequest analyse aussi les données de formulaire HTTP entrantes et les stockent sous forme de paramètres de servlet. Le serveur passe un objet HttpServletRequest à la méthode de service d'un HttpServlet.

Certaines méthodes de cette interface ont souffert d'incohérences entre la documentation et l'implémentation. Les contradictions sont signalées où c'est possible.

Déclaration d'interface

```
public interface HttpServletRequest extends javax.servlet.ServletRequest {
  // Méthodes.
  public abstract String getAuthType();
  public abstract String getContextPath();                // Nouvelle en 2.2
  public abstract Cookie[] getCookies();                  // Nouvelle en 2.0
  public abstract long getDateHeader(String name);
  public abstract String getHeader(String name);
  public abstract Enumeration getHeaderNames();
  public abstract Enumeration getHeaders(String name);    // Nouvelle en 2.2
  public abstract int getIntHeader(String name);
  public abstract String getMethod();
  public abstract String getPathInfo();
  public abstract String getPathTranslated();
  public abstract String getQueryString();
```

```
    public abstract String getRemoteUser();
    public abstract String getRequestedSessionId();        // Nouvelle en 2.0
    public abstract String getRequestURI();
    public abstract String getServletPath();
    public abstract HttpSession getSession();              // Nouvelle en 2.1
    public abstract HttpSession getSession(boolean create);// Nouvelle en 2.0
    public abstract
      java.security.Principal getUserPrincipal();          // Nouvelle en 2.2
    public abstract
      boolean isRequestedSessionIdFromCookie();            // Nouvelle en 2.0
    public abstract boolean isRequestedSessionIdFromUrl(); // Désapprouvée
    public abstract boolean isRequestedSessionIdFromURL(); // Nouvelle en 2.1
    public abstract boolean isRequestedSessionIdValid();   // Nouvelle en 2.0
    public abstract boolean isUserInRole(String role);     // Nouvelle en 2.2
}
```

Méthodes

`getAuthType()`

```
public abstract String getAuthType()
```

Description

Retourne le schéma d'authentification de la servlet ou `null` si la servlet n'était pas protégée par un mécanisme de contrôle d'accès. Les schémas possibles sont `BASIC`, `DIGEST`, `FORM` et `CLIENT-CERT`. Identique à la variable CGI `AUTH_TYPE`.

`getContextPath()`

```
public abstract String getContextPath()
```

Description

Retourne la partie de l'URI de requête qui indique le contexte (l'application web) de la requête. Le chemin de contexte vient toujours au début de l'URI de requête. Il commence par un caractère / mais ne se termine pas par le caractère / et il est retourné tel quel sans passer par le décodage d'URL. Pour les servlets présentes dans le contexte par défaut (racine), cette méthode renvoie une chaîne vide. Elle a été introduite dans l'API Servlet 2.2.

`getCookies()`

```
public abstract Cookie[] getCookies()
```

Description

Retourne un tableau d'objets `Cookie` qui contient tous les cookies envoyés par le navigateur dans la requête ou `null` si aucun cookie n'a été envoyé. Cette méthode a été introduite dans l'API Servlet 2.0.

`getDateHeader()`

```
public abstract long getDateHeader(String name)
```

Description

Retourne la valeur de l'en-tête indiqué sous forme d'un `long` représentant une `Date` (le nombre de millisecondes écoulées depuis le 1er janvier 1970 GMT à mi-

nuit) ou -1 si l'en-tête n'a pas été envoyé dans la requête. Le nom est sensible à la casse. Lance une `IllegalArgumentException` lorsqu'elle est appelée sur un en-tête dont la valeur ne peut pas être convertie en un `Date`. Cette méthode est utile pour traiter des en-têtes comme `Last-Modified` et `If-Modified-Since`.

`getHeader()`

```
public abstract String getHeader(String name)
```

Description

Retourne la valeur de l'en-tête indiqué sous forme d'un `String` ou `null` si l'en-tête ne faisait pas partie de la requête. Le nom est sensible à la casse. Cette méthode peut retrouver tous les types d'en-tête.

`getHeaderNames()`

```
public abstract Enumeration getHeaderNames()
```

Description

Retourne le nom de tous les en-têtes auxquels une servlet a accès sous forme d'un `Enumeration` d'objets `String` ou d'un `Enumeration` vide s'il n'y a pas d'en-têtes. Certaines implémentations de servlets peuvent ne pas autoriser l'accès aux en-têtes de cette façon, auquel cas cette méthode retourne `null`.

`getHeaders()`

```
public abstract Enumeration getHeaders(String name)
```

Description

Retourne toutes les valeurs de l'en-tête indiqué sous forme d'un `Enumeration` d'objets `String`. Certains en-têtes, comme `Accept-Language`, peuvent être envoyés plusieurs fois par le client avec une valeur différentes. Cette méthode permet de retrouver toutes ces valeurs. Si la requête n'inclut aucun en-tête du nom indiqué, cette méthode retourne un `Enumeration` vide. Le nom est sensible à la casse. Cette méthode peut retrouver tout type d'en-tête. Elle a été introduite dans l'API Servlet 2.2.

`getIntHeader()`

```
public abstract int getIntHeader(String name)
```

Description

Retourne la valeur de l'en-tête indiqué sous forme d'un `int` ou -1 si l'en-tête n'a pas été envoyé dans la requête. Le nom est sensible à la casse. Lance une `NumberFormatException` lorsqu'elle est appelée sur un en-tête dont la valeur ne peut être convertie en un `int`.

`getMethod()`

```
public abstract String getMethod()
```

Description

Retourne la méthode HTTP utilisée pour effectuer la requête. Des exemples de méthodes incluent `GET`, `POST` et `HEAD`. Identique à la variable CGI `REQUEST_METHOD`. L'implémentation de `HttpServlet` de `service()` utilise cette méthode pour la distribution des requêtes.

getPathInfo()

```
public abstract String getPathInfo()
```

Description

Retourne l'information de chemin supplémentaire associée à la requête ou null si aucune n'a été fournie. Le chemin est décodé avant d'être retourné. Identique à la variable CGI PATH_INFO.

getPathTranslated()

```
public abstract String getPathTranslated()
```

Description

Retourne l'information de chemin supplémentaire traduite en un chemin sur le système de fichiers ou null si aucune information n'a été fournie ou si le conteneur de servlets ne peut créer un chemin de fichier valide (par exemple dans le cas où le fichier se trouve sur un système de fichiers distant ou uniquement dans une archive *.war*). Le chemin retourné ne pointe pas nécessairement sur un fichier ou un répertoire existant. Le chemin est décodé avant d'être retourné. Identique à la variable CGI PATH_TRANSLATED ainsi qu'à `getServletContext().getRealPath(req.getPathInfo())`.

getQueryString()

```
public abstract String getQueryString()
```

Description

Retourne la chaîne d'interrogation de l'URL de la requête. Identique à la variable CGI QUERY_STRING. Puisque HttpServletRequest analyse cette chaîne pour que les paramètres de la servlet soient disponibles *via* `getParameter()`, la plupart des servlets peuvent ignorer cette méthode.

getRemoteUser()

```
public abstract String getRemoteUser()
```

Description

Retourne le nom de l'utilisateur effectuant la requête sous forme d'un String ou null si l'accès à la servlet n'était pas restreint. Identique à la variable CGI REMOTE_USER. En général, il faut que l'utilisateur se soit connecté en utilisant une authentification HTTP. Il n'y a pas de méthode comparable pour avoir accès directement au mot de passe de l'utilisateur distant.

getRequestedSessionId()

```
public abstract String getRequestedSessionId()
```

Description

Cette méthode retourne l'identifiant de session spécifié par le client. Il se peut que ce ne soit pas l'identifiant de session réel en cours d'utilisation. Par exemple, si la session a expiré avant que la requête n'arrive, le serveur crée un nouvel identifiant de session et l'utilise à la place. Cette méthode a été introduite dans l'API Servlet 2.0.

`getRequestURI()`

```
public abstract String getRequestURI()
```

Description

Retourne l'URI (*Universal Resource Identifier*) de la requête. C'est la ressource demandée par le client dans la première ligne de sa requête HTTP, c'est-à-dire tout ce qui se trouve après le protocole et avant la chaîne d'interrogation. Pour les servlets HTTP normales, l'URI de requête est l'URL de requête moins le schéma, l'hôte, le port et la chaîne d'interrogation mais avec l'information de chemin supplémentaire. Le chemin est décodé avant d'être retourné. Des versions précédentes de l'API Servlet ont défini et implémenté cette méthode de différentes façons. Lorsque vous écrivez du code dépendant de cette méthode, assurez-vous que vous savez ce que vous obtenez.

`getServletPath()`

```
public abstract String getServletPath()
```

Description

Retourne la partie de l'URI qui fait référence à la servlet. Le chemin est décodé avant d'être retourné et il n'inclut pas l'information de chemin supplémentaire ni la chaîne d'interrogation. Pour les correspondances d'extension de fichiers, le chemin inclut l'extension. Identique à la variable CGI `SCRIPT_NAME`.

`getSession()`

```
public abstract HttpSession getSession()
public abstract HttpSession getSession(boolean create)
```

Description

Retourne la session courante associée à l'utilisateur effectuant la requête. Si l'utilisateur n'a pas de session valide, cette méthode en crée une si `create` vaut `true` ou retourne `null` sinon. Dans la version sans argument, `create` vaut `true` par défaut. Pour être sûr que la session est correctement maintenue, cette méthode doit être appelée au moins une fois avant d'écrire toute sortie sur la réponse. Les servlets n'utilisant pas le suivi de session peuvent ignorer cette méthode. Cette méthode a été introduite dans l'API Servlet 2.0. La version sans argument a été introduite dans l'API Servlet 2.1.

`getUserPrincipal()`

```
public abstract java.security.Principal getUserPrincipal()
```

Description

Retourne un objet `java.security.Principal` contenant le nom de l'utilisateur courant authentifié. Si l'utilisateur n'a pas été authentifié, la méthode retourne `null`. Cette méthode a été introduite dans l'API Servlet 2.2.

`isRequestedSessionIdFromCookie()`

```
public abstract boolean isRequestedSessionIdFromCookie()
```

Description

Retourne `true` si le client a soumis un identifiant de session *via* un cookie, sinon `false`. Cette méthode a été introduite dans l'API Servlet 2.0.

isRequestedSessionIdFromURL()

```
public abstract boolean isRequestedSessionIdFromUrl()
public abstract boolean isRequestedSessionIdFromURL()
```

Description

Retourne true si le client a soumis un identifiant de session *via* une URL réécrite, sinon false. La méthode isRequestedSessionIdFromUrl() a été introduite dans l'API Servlet 2.0, puis désapprouvée dans l'API Servlet 2.1 avec l'introduction de la méthode isRequestedSessionIdFromURL() au nom plus standard.

isRequestedSessionIdValid()

```
public abstract boolean isRequestedSessionIdValid()
```

Description

Retourne true si la session demandée par le client est valide et si elle est la session en cours d'utilisation. Pour des sessions nouvelles ou expirées, elle retourne false. Cette méthode a été introduite dans l'API Servlet 2.0

isUserInRole()

```
public abstract boolean isUserInRole(String role)
```

Description

Retourne un boolean indiquant si l'utilisateur authentifié fait partie du « rôle » logique donné. Les rôles et les appartenances aux rôles sont définies à l'aide des descripteurs de déploiement, et les rôles sont associés aux utilisateurs et aux groupes à l'aide des outils d'administration du serveur. Si l'utilisateur n'a pas été authentifié, cette méthode retourne toujours false. Elle a été introduite dans l'API Servlet 2.2.

HttpServletResponse

Synopsis

Nom de l'interface :	javax.servlet.http.HttpServletResponse
Interface mère :	javax.servlet.ServletResponse
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

HttpServletResponse étend la classe ServletResponse pour permettre la manipulation des données spécifiques au protocole HTTP, y compris les en-têtes et les codes d'état de la réponse. Elle définit aussi une série de constantes qui représentent divers codes d'état HTPP et elle inclut des fonctions facilitant les opérations de suivi de session.

Déclaration d'interface

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {
  // Constantes.
  public static final int SC_ACCEPTED;
  public static final int SC_BAD_GATEWAY;
  public static final int SC_BAD_REQUEST;
  public static final int SC_CONFLICT;
  public static final int SC_CONTINUE;                    // Nouvelle en 2.0
  public static final int SC_CREATED;
  public static final int SC_EXPECTATION_FAILED;          // Nouvelle en 2.2
  public static final int SC_FORBIDDEN;
  public static final int SC_GATEWAY_TIMEOUT;             // Nouvelle en 2.0
  public static final int SC_GONE;                        // Nouvelle en 2.0
  public static final int SC_HTTP_VERSION_NOT_SUPPORTED; // Nouvelle en 2.0
  public static final int SC_INTERNAL_SERVER_ERROR;
  public static final int SC_LENGTH_REQUIRED;             // Nouvelle en 2.0
  public static final int SC_METHOD_NOT_ALLOWED;          // Nouvelle en 2.0
  public static final int SC_MOVED_PERMANENTLY;
  public static final int SC_MOVED_TEMPORARILY;
  public static final int SC_MULTIPLE_CHOICES;            // Nouvelle en 2.0
  public static final int SC_NO_CONTENT;
  public static final int SC_NON_AUTHORITATIVE_INFORMATION; // Nouv. en 2.0
  public static final int SC_NOT_ACCEPTABLE;              // Nouvelle en 2.0
  public static final int SC_NOT_FOUND;
  public static final int SC_NOT_IMPLEMENTED;
  public static final int SC_NOT_MODIFIED;
  public static final int SC_OK;
  public static final int SC_PARTIAL_CONTENT;             // Nouvelle en 2.0
  public static final int SC_PAYMENT_REQUIRED;            // Nouvelle en 2.0
  public static final int SC_PRECONDITION_FAILED;         // Nouvelle en 2.0
  public static final int SC_PROXY_AUTHENTICATION_REQUIRED; // Nouv. en 2.0
  public static final int SC_REQUEST_ENTITY_TOO_LARGE;    // Nouvelle en 2.0
  public static final int SC_REQUEST_TIMEOUT;             // Nouvelle en 2.0
  public static final int SC_REQUEST_URI_TOO_LONG;        // Nouvelle en 2.0
  public static final int SC_REQUESTED_RANGE_NOT_SATISFIABLE;// Nouv. en 2.2
  public static final int SC_RESET_CONTENT;               // Nouvelle en 2.0
  public static final int SC_SEE_OTHER;                   // Nouvelle en 2.0
  public static final int SC_SERVICE_UNAVAILABLE;
  public static final int SC_SWITCHING_PROTOCOLS;         // Nouvelle en 2.0
  public static final int SC_UNAUTHORIZED;
  public static final int SC_UNSUPPORTED_MEDIA_TYPE;      // Nouvelle en 2.0
  public static final int SC_USE_PROXY;                   // Nouvelle en 2.0

  // Méthodes.
  public abstract void addCookie(Cookie cookie);          // Nouvelle en 2.0
  public abstract
    void addDateHeader(String name, long date);           // Nouvelle en 2.2
  public abstract
    void addHeader(String name, String value);            // Nouvelle en 2.2
  public abstract
    void addIntHeader(String name, int value);            // Nouvelle en 2.2
```

```
    public abstract boolean containsHeader(String name);
    public abstract String encodeRedirectUrl(String url);  // Désapprouvée
    public abstract String encodeRedirectURL(String url);  // Nouvelle en 2.1
    public abstract String encodeUrl(String url);          // Désapprouvée
    public abstract String encodeURL(String url);          // Nouvelle en 2.1
    public abstract void sendError(int sc)
      throws IOException, IllegalStateException;
    public abstract void sendError(int sc, String msg)
      throws IOException, IllegalStateException;
    public abstract void sendRedirect(String location)
      throws IOException, IllegalStateException;
    public abstract void setDateHeader(String name, long date);
    public abstract void setHeader(String name, String value);
    public abstract void setIntHeader(String name, int value);
    public abstract void setStatus(int sc);
    public abstract void setStatus(int sc, String sm);     // Désapprouvée
}
```

Constantes

L'annexe D, *Codes d'état HTTP*, contient une description complète de tous les codes d'état `SC_XXX`.

Méthodes

`addCookie()`

```
public abstract void addCookie(Cookie cookie)
```

Description

Ajoute le cookie indiqué à la réponse. D'autres cookies peuvent être ajoutés par des appels répétés à `addCookie()`. Les cookies étant envoyés en utilisant des entêtes HTTP, ils doivent être ajoutés à la réponse avant qu'elle soit validée. Les navigateurs ont l'obligation de n'accepter que 20 cookies par site, 300 au total par utilisateur et ils peuvent limiter la taille de chaque cookie à 4 096 octets.

`addDateHeader()`

```
public abstract void addDateHeader(String name, long date)
```

Description

Ajoute un en-tête de nom et de valeur indiqués. La méthode accepte la date sous forme d'un `long` qui représente le nombre de millisecondes écoulées depuis le 1er janvier 1970 GMT à minuit. Cette méthode permet de donner plusieurs valeurs aux en-têtes de réponse. Elle a été introduite dans l'API Servlet 2.2.

`addHeader()`

```
public abstract void addHeader(String name, String value)
```

Description

Ajoute un en-tête de nom et de valeur indiqués. Cette méthode permet de donner plusieurs valeurs aux en-têtes de réponse. Elle a été introduite dans l'API Servlet 2.2.

addIntHeader()

```
public abstract void addIntHeader(String name, int value)
```

Description

Ajoute un en-tête de nom et de valeur int indiqués. La méthode accepte la valeur sous forme d'un long qui représente le nombre de millisecondes écoulées depuis le 1er janvier 1970 GMT à minuit. Cette méthode permet de donner plusieurs valeurs aux en-têtes de réponse. Elle a été introduite dans l'API Servlet 2.2.

containsHeader()

```
public abstract boolean containsHeader(String name)
```

Description

Retourne true si l'en-tête spécifié a déjà été défini, sinon false.

encodeRedirectURL()

```
public abstract String encodeRedirectUrl(String url)
public abstract String encodeRedirectURL(String url)
```

Description

Retourne l'URL spécifiée codée (réécrite) de façon à inclure l'identifiant de session. Si le codage n'est pas nécessaire ou s'il n'est pas supporté, la méthode laisse l'URL telle quelle. Les règles utilisées pour décider du moment et de la façon de coder une URL sont spécifiques au serveur. Cette méthode peut utiliser des règles différentes de encodeURL(). Pour activer le suivi de session, toute URL passée à la méthode sendRedirect() doit passer par cette méthode. La méthode encodeRedirectUrl() a été introduite dans l'API Servlet 2.0, puis désapprouvée dans l'API Servlet 2.1 avec l'introduction de la méthode encodeRedirectURL() au nom plus standard.

encodeURL()

```
public abstract String encodeUrl(String url)
public abstract String encodeURL(String url)
```

Description

Retourne l'URL spécifiée codée (réécrite) de façon à inclure l'identifiant de session. Si le codage n'est pas nécessaire ou s'il n'est pas supporté, la méthode laisse l'URL telle quelle. Les règles utilisées pour décider du moment et de la façon de coder une URL sont spécifiques au serveur. Pour activer le suivi de session, toute URL émise par une servlet doit passer par cette méthode. La méthode encodeUrl() a été introduite dans l'API Servlet 2.0, puis désapprouvée dans l'API Servlet 2.1 avec l'introduction de la méthode encodeURL() au nom plus standard.

sendError()

```
public abstract void sendError(int sc)
  throws IOException, IllegalStateException
public abstract void sendError(int sc, String msg)
  throws IOException, IllegalStateException
```

Description

Ces méthodes sont similaires à setStatus(), excepté qu'elles sont utilisées lorsque le code d'état indique une erreur pendant le traitement de la requête et lors-

que la servlet veut que le serveur génère une page d'erreur appropriée. Cette méthode doit être appelée avant que la réponse ne soit validée, sinon elle lance une `IllegalStateException`. Cette méthode réinitialise implicitement le tampon de réponse avant de générer la page d'erreur. Les en-têtes définis avant l'appel à `sendError()` sont conservés.

`sendRedirect()`

```
public abstract void sendRedirect(String location)
  throws IOException, IllegalStateException
```

Description

Redirige la réponse vers l'endroit spécifié, en définissant automatiquement le code d'état et l'en-tête `Location`. L'implémentation par défaut écrit aussi un petit corps de réponse qui contient un lien hypertexte vers le nouvel emplacement, pour les navigateurs sans possibilité de redirection. Par conséquent, vous ne devez pas écrire votre propre corps de réponse lorsque vous employez cette méthode. La spécification HTTP stipule que toutes les URL de redirection doivent être absolues ; cependant, à partir de l'API Servlet 2.2, cette méthode accepte des URL relatives, le serveur convertissant l'URL en sa forme absolue (il ajoute automatiquement le protocole, le serveur et le port courants, mais pas le chemin de contexte ; faites-le vous-mêmes si nécessaire) avant de l'envoyer au client. Cette méthode doit être appelée avant que la réponse ne soit validée, sinon elle lance une `IllegalStateException`. Elle réinitialise implicitement le tampon de réponse avant de générer la page de redirection. Les en-têtes définis avant l'appel à `sendRedirect()` sont conservés.

`setDateHeader()`

```
public abstract void setDateHeader(String name, long date)
```

Description

Fixe la valeur de l'en-tête indiqué sous forme d'un `String` à la date donnée. La méthode accepte la date sous forme d'un `long` qui représente le nombre de millisecondes écoulées depuis le 1er janvier 1970 GMT à minuit. Si l'en-tête a déjà été fixé, la nouvelle valeur écrase la précédente.

`setHeader()`

```
public abstract void setHeader(String name, String value)
```

Description

Fixe la valeur de l'en-tête indiqué sous forme d'un `String`. Le nom est sensible à la casse (comme avec toutes les méthodes liées aux en-têtes). Si l'en-tête a déjà été fixé, la nouvelle valeur écrase la précédente. Cette méthode peut fixer tout type d'en-tête. La valeur des en-têtes doit toujours être fixée avant que la réponse ne soit validée.

`setIntHeader()`

```
public abstract void setIntHeader(String name, int value)
```

Description

Fixe la valeur de l'en-tête indiqué sous forme d'un `int`. Si l'en-tête a déjà été fixé, la nouvelle valeur écrase la précédente.

`setStatus()`

```
public abstract void setStatus(int sc)
public abstract void setStatus(int sc, String sm)
```

Description

Définit le code d'état HTTP. Le code peut être spécifié en utilisant une valeur numérique ou par les codes `SC_XXX` définis par `HttpServletResponse`. Vous pouvez spécifier un message d'erreur personnalisé dans le second paramètre ; cependant, il ne faut pas le faire et la méthode acceptant un `String` a été désapprouvée dans l'API Servlet 2.1. Le code d'état doit être fixé avant que la réponse ne soit validée, sinon l'appel est ignoré.

HttpSession

Synopsis

Nom de l'interface :	`javax.servlet.http.HttpSession`
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 2.0 et ultérieure

Description

L'interface `HttpSession` offre un mécanisme de conservation temporaire des informations concernant les visiteurs consultant le site web. Pour une introduction détaillée au suivi de session, reportez-vous au chapitre 7. L'interface `HttpSession` permet aux servlets de voir et de manipuler les informations spécifiques à une session, comme l'heure de création et l'identifiant de session unique. Elle inclut aussi des méthodes permettant de lier des objets à la session afin de les retrouver plus tard, par exemple pour qu'une application de caddie virtuel, ou d'autres applications, conserve des données entre les requêtes des clients.

Une servlet obtient un objet `HttpSession` à partir de la méthode `getSession()` de `HttpServletRequest`. Le comportement de la session, comme le temps de pause avant sa destruction, peut être configuré par programmation et par le descripteur de déploiement de l'application.

Pour les sessions qui se trouvent dans une application web non distribuée, tout objet peut être lié à la session. Les objets implémentant `java.io.Serializable` peuvent être écrits sur le disque afin d'économiser la mémoire et de persister entre les redémarrages du serveur.

Pour les sessions qui se trouvent dans une application web distribuée, les objets placés dans la session *doivent* implémenter `java.io.Serializable`. Le serveur lance une `IllegalArgumentException` si cette contrainte n'est pas respectée. Les serveurs utilisent l'*affinité de session* pour gérer les sessions dans un environnement distribué comprenant plusieurs serveurs. Cela signifie que toutes les requêtes faisant partie d'une même ses-

sion d'un utilisateur particulier sont gérées par une seule JVM à la fois. Cela permet d'éliminer le besoin de réplication des informations de session sur tous les serveurs. La responsabilité d'une session peut passer d'un serveur à un autre entre les requêtes de l'utilisateur. Pour permettre le déplacement d'une session, tous les objets de la session doivent être `Serializable`.

Déclaration d'interface

```
public interface HttpSession {
  // Méthodes.
  // La plupart des méthodes lancent une IllegalStateException.
  public abstract Object getAttribute(String name);       // Nouvelle en 2.2
  public abstract Enumeration getAttributeNames();        // Nouvelle en 2.2
  public abstract long getCreationTime();
  public abstract String getId();
  public abstract long getLastAccessedTime();
  public abstract int getMaxInactiveInterval();           // Nouvelle en 2.1
  public abstract HttpSessionContext getSessionContext();// Désapprouvée
  public abstract Object getValue(String name);           // Désapprouvée
  public abstract String[] getValueNames();               // Désapprouvée
  public abstract void invalidate();
  public abstract boolean isNew();
  public abstract
    void putValue(String name, Object value);             // Désapprouvée
  public abstract void removeAttribute(String name);      // Nouvelle en 2.2
  public abstract void removeValue(String name);          // Désapprouvée
  public abstract
    void setAttribute(String name, Object value);         // Nouvelle en 2.2
  public abstract void setMaxInactiveInterval(int secs); // Nouvelle en 2.1
}
```

Méthodes

`getAttribute()`

```
public abstract Object getAttribute(String name)
  throws IllegalStateException
```

Description

Retourne l'objet lié à la session sous le nom indiqué ou `null` si la liaison n'existe pas. Lance une `IllegalStateException` si la session est invalide.

`getAttributeNames()`

```
public abstract Enumeration getAttributeNames()
  throws IllegalStateException
```

Description

Retourne les noms de tous les objets liés à la session sous forme d'un `Enumeration` d'objets `String` ou un `Enumeration` vide s'il n'y a pas de liaison. Lance une `IllegalStateException` si la session est invalide.

getCreationTime()

```
public abstract long getCreationTime()
  throws IllegalStateException
```

Description

Retourne la date de création de la session, sous forme d'un long représentant le nombre de millisecondes depuis le 1er janvier 1970 GMT à minuit. Lance une IllegalStateException si la session est invalide.

getId()

```
public abstract String getId()
  throws IllegalStateException
```

Description

Retourne l'identifiant String unique attribué à cette session. La structure de l'identifiant dépend de l'implémentation ; il doit simplement être difficile à deviner. Par exemple, un identifiant de Tomcat pourrait être awj4qyhsn2. Lance une IllegalStateException si la session est invalide.

getLastAccessedTime()

```
public abstract long getLastAccessedTime()
  throws IllegalStateException
```

Description

Retourne la date à laquelle le client a envoyé pour la dernière fois une requête associée à cette session, sous forme d'un long représentant le nombre de millisecondes depuis le 1er janvier 1970 GMT à minuit. Lance une IllegalStateException si la session est invalide.

getMaxInactiveInterval()

```
public abstract int getMaxInactiveInterval()
  throws IllegalStateException
```

Description

Retourne la durée maximale, en secondes, entre des requêtes du client avant que le conteneur de servlets invalide la session. Une valeur négative indique que la session ne doit jamais expirer. La durée par défaut est définie dans le descripteur de déploiement de l'application web. Lance une IllegalStateException si la session est invalide. Cette méthode a été introduite dans l'API Servlet 2.1.

getSessionContext()

```
public abstract HttpSessionContext getSessionContext()
```

Description

À partir de l'API Servlet 2.1, retourne un contexte vide pour des raisons de sécurité. Cette méthode a été désapprouvée à partir de l'API Servlet 2.1. Voyez HttpSessionContext pour plus d'informations.

getValue()

```
public abstract Object getValue(String name)
  throws IllegalStateException
```

Description

Retourne l'objet lié à la session sous le nom spécifié ou null s'il la liaison n'existe pas. Lance une IllegalStateException si la session est invalide. Désapprouvée à partir de l'API Servlet 2.2 en faveur de getAttribute().

getValueNames()

```
public abstract String[] getValueNames()
  throws IllegalStateException
```

Description

Retourne un tableau contenant le nom de tous les objets liés à cette session ou un tableau vide (de longueur zéro) s'il n'y a aucune liaison. Lance une IllegalStateException si la session est invalide. Cette méthode a été désapprouvée à partir de l'API Servlet 2.2 en faveur de getAttributeNames().

invalidate()

```
public abstract void invalidate()
  throws IllegalStateException
```

Description

Provoque l'invalidation immédiate de la session. Tous les objets stockés dans la session sont déliés. Lance une IllegalStateException si la session est déjà invalide.

isNew()

```
public abstract boolean isNew()
  throws IllegalStateException
```

Description

Indique si la session est nouvelle ou non. Une session est considérée nouvelle si elle a été créée par le serveur mais que le client n'a pas encore confirmé qu'il se joignait à la session. Par exemple, si un serveur supporte uniquement les sessions à base de cookie et qu'un client a complètement désactivé leur utilisation, les appels à getSession() indiquent toujours que la session est nouvelle. Lance une IllegalStateException si la session est invalide.

putValue()

```
public abstract void putValue(String name, Object value)
  throws IllegalStateException
```

Description

Lie la valeur de l'objet spécifiée sous le nom spécifié dans la session. Toute liaison existante de même nom est remplacée. Lance une IllegalStateException si la session est invalide. Cette méthode a été désapprouvée à partir de l'API Servlet 2.2 en faveur de setAttribute().

removeAttribute()

```
public abstract void removeAttribute(String name)
  throws IllegalStateException
```

Description

Supprime l'objet lié sous le nom indiqué ou ne fait rien si la liaison n'existe pas. Lance une `IllegalStateException` si la session est invalide.

`removeValue()`

```
public abstract void removeValue(String name)
  throws IllegalStateException
```

Description

Supprime l'objet lié sous le nom indiqué ou ne fait rien si la liaison n'existe pas. Lance une `IllegalStateException` si la session est invalide. Cette méthode a été désapprouvée à partir de l'API Servlet 2.2 en faveur de `removeAttribute()`.

`setAttribute()`

```
public abstract void setAttribute(String name, Object value)
  throws IllegalStateException
```

Description

Lie la valeur de l'objet spécifiée sous le nom spécifié dans la session. Toute liaison existante de même nom est remplacée. Lance une `IllegalStateException` si la session est invalide. Cette méthode a été introduite dans l'API Servlet 2.2.

`setMaxInactiveInterval()`

```
public abstract void setMaxInactiveInterval(int secs)
```

Description

Spécifie la durée maximale, en secondes, entre des requêtes du client avant que le conteneur de servlets invalide la session. Une valeur négative indique que la session ne doit jamais expirer. La durée par défaut est définie dans le descripteur de déploiement de l'application web. Cette méthode a été introduite dans l'API Servlet 2.1.

HttpSessionBindingEvent

Synopsis

Nom de classe :	`javax.servlet.http.HttpSessionBindingEvent`
Classe mère :	`java.util.EventObject`
Sous-classes directes :	aucune
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 2.0 et ultérieure

Description

Un `HttpSessionBindingEvent` est passé à un `HttpSessionBindingListener` lorsque l'objet auditeur est lié ou délié de la session.

Résumé

```
public class HttpSessionBindingEvent extends java.util.EventObject {
  // Constructeurs.
  public HttpSessionBindingEvent(HttpSession session, String name);

  // Méthodes d'instance.
  public String getName();
  public HttpSession getSession();
}
```

Constructeurs

HttpSessionBindingEvent()

```
public HttpSessionBindingEvent(HttpSession session, String name)
```

Description

Construit un nouveau `HttpSessionBindingEvent` en utilisant la session en cours de liaison et le nom en cours d'attribution à l'objet (c'est le même nom que celui passé à la méthode `setAttribute()` de `HttpSession`). Les programmeurs de servlets ne doivent jamais utiliser ce constructeur.

Méthodes d'instance

getName()

```
public String getName()
```

Description

Retourne le nom attribué à cet objet dans la session.

getSession()

```
public HttpSession getSession()
```

Description

Retourne la session à laquelle cet objet est en train d'être lié ou délié.

HttpSessionBindingListener

Synopsis

Nom de l'interface :	`javax.servlet.http.HttpSessionBindingListener`
Interface mère :	`java.util.EventListener`
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 2.0 et ultérieure

Description

Un objet implémentant HttpSessionBindingListener reçoit, *via* des appels à valueBound() et valueUnbound(), des notifications l'informant lorsqu'il est lié ou délié d'un HttpSession. En autres choses, cette interface autorise une libération ordonnée des ressources spécifiques à une session, comme les connexions à une base de données. Notez que dans un environnement distribué, l'appel à valueBound() peut se faire au sein d'une JVM différente de celle qui appelle valueUnbound().

Déclaration d'interface

```
public interface HttpSessionBindingListener
      extends java.util.EventListener {
  // Méthodes.
  public abstract void valueBound(HttpSessionBindingEvent event);
  public abstract void valueUnbound(HttpSessionBindingEvent event);
}
```

Méthodes

valueBound()

```
public abstract void valueBound(HttpSessionBindingEvent event)
```

Description

Appelée lorsque l'auditeur est lié à une session.

valueUnbound()

```
public abstract void valueUnbound(HttpSessionBindingEvent event)
```

Description

Appelée lorsque l'auditeur est délié d'une session (y compris à la destruction de la session).

HttpSessionContext

Synopsis

Nom de l'interface :	javax.servlet.http.HttpSessionContext
Interface mère :	aucune
Sous-interfaces directes :	aucune
Implémentée par :	aucune
Disponibilité :	API Servlet 2.0 et ultérieure ; désapprouvée dans l'API Servlet 2.1

Description

HttpSessionContext est désapprouvée à partir de l'API Servlet 2.1. Dans les versions précédentes, cette classe fournit l'accès à tout ce qui concerne les sessions actives dans le conteneur se servlets. Cet accès est un trou de sécurité potentiel que les servlets peuvent

utiliser pour afficher les identifiants de session du contexte, information que des clients peu scrupuleux peuvent exploiter pour s'immiscer dans une session d'un autre client. Puisque les accès à toutes les sessions ne sont quasiment jamais nécessaire, cette classe a été désapprouvée afin de garantir une meilleure sécurité.

Déclaration d'interface

```
public interface HttpSessionContext {
  // Méthodes.
  public abstract Enumeration getIds();                          // Désapprouvée
  public abstract HttpSession getSession(String sessionId); // Désapprouvée
}
```

Méthodes

`getIds()`

```
public abstract Enumeration getIds()
```

Description

Désapprouvée à partir de l'API Servlet 2.1. Dans l'API Servlet 2.0, cette méthode retourne un `Enumeration` contenant les identifiants de toutes les sessions actuellement valides dans le contexte, ou un `Enumeration` vide s'il n'y a pas de session valide. Les identifiants de session retournés par `getIds()` doivent être conservés secrets par le serveur car tout client connaissant l'identifiant de session d'un autre client peut, grâce à un cookie ou une URL fabriqués, se joindre à la session de ce client.

`getSession()`

```
public abstract HttpSession getSession(String sessionId)
```

Description

Désapprouvée à partir de l'API Servlet 2.1. Dans l'API Servlet 2.0, cette méthode retourne la session associée à l'identifiant de session spécifié. Une liste d'identifiants de session valide peut être obtenue par la méthode `getIds()`.

HttpUtils

Synopsis

Nom de classe :	`javax.servlet.http.HttpUtils`
Classe mère :	`java.lang.Object`
Sous-classes directes :	aucune
Interfaces implémentées :	aucune
Disponibilité :	API Servlet 1.0 et ultérieure

Description

Un objet conteneur pour un ensemble de méthodes orientées HTTP potentiellement utiles.

Résumé

```
public class HttpUtils {
  // Constructeurs.
  public HttpUtils();

  // Méthodes de classe.
  public static StringBuffer getRequestURL(HttpServletRequest req);
  public static Hashtable parsePostData(int len, ServletInputStream in);
  public static Hashtable parseQueryString(String s);
}
```

Constructeurs

`HttpUtils()`

```
public HttpUtils()
```

Description

Le constructeur par défaut ne fait rien.

Méthodes de classe

`getRequestURL()`

```
public static StringBuffer getRequestURL(HttpServletRequest req)
```

Description

Reconstitue l'URL demandée en fonction des informations disponibles dans l'objet `HttpServletRequest`. Retourne un `StringBuffer` qui inclut le schéma, le nom du serveur, le port du serveur et l'information de chemin supplémentaire, mais pas la chaîne d'interrogation. L'URL reconstituée doit être très similaire à celle utilisée par le client. Cette méthode peut être utilisée pour rapporter des erreurs, pour la redirection et pour la création d'URL. Pour les applications ayant besoin d'identifier de façon unique des servlets, la méthode `getRequestURI()` de `HttpServletRequest` est généralement un meilleur choix.

`parsePostData()`

```
public static Hashtable parsePostData(int len, ServletInputStream in)
```

Description

Analyse `len` caractères de données venant d'un `ServletInputStream` (habituellement envoyé dans une opération POST). Lance une `IllegalArgumentException` si les données en paramètre sont invalides. La plupart des servlets utilisent `getParameterNames()`, `getParameter()` et `getParameterValues()` au lieu de cette méthode.

`parseQueryString()`

```
public static Hashtable parseQueryString(String s)
```

Description

Retourne un `Hashtable` où les clés de hachage sont les noms des paramètres pris dans la chaîne d'interrogation et chaque valeur de hachage est un tableau de

`String` qui contient la ou les valeurs décodées des paramètres. Lance une `IllegalArgumentException` si la chaîne d'interrogation est invalide. La plupart des servlets utilisent plutôt `getParameterNames()`, `getParameter()` et `getParameterValues()`. Il n'est pas sûr d'utiliser les deux.

C

Descripteur de déploiement, référence de la DTD

Un fichier de DTD (*Document Type Definition*) décrit ce qui représente un contenu valide dans un fichier XML. Cette annexe détaille la DTD officielle du descripteur de déploiement d'une application web pour l'API Servlet 2.2. Les fichiers de DTD ne sont pas du XML, mais leur syntaxe est facile : chaque balise `<!ELEMENT>` définit entre parenthèses les éléments enfant qu'elle peut avoir, combien elle peut avoir de chaque enfant et dans quel ordre. Un point d'interrogation (?) après le nom d'un fils indique que celui-ci est optionnel (*0 – 1*), un astérisque (*) indique que le fils peut apparaître un nombre quelconque de fois (*0 – n*), un symbole plus (+) signifie qu'un certain nombre d'enfants doivent être présents (*1 – n*) et aucun caractère signifie que le fils est obligatoire (*1*). La syntaxe `(x|y)` signifie x ou y. Les fils d'un élément doivent être donnés dans l'ordre dans lequel ils apparaissent entre les parenthèses.

Les *figures C-1* à *C-4* montre graphiquement la structure des éléments ; l'*exemple C-1* montre la DTD elle-même. Chaque balise `<!ATTLIST>` contrôle les attributs autorisés pour chaque élément. Dans cette DTD, la balise est utilisée uniquement pour fournir les valeurs par défaut de `id`. Le reste de cette annexe constitue la référence des éléments de cette DTD.

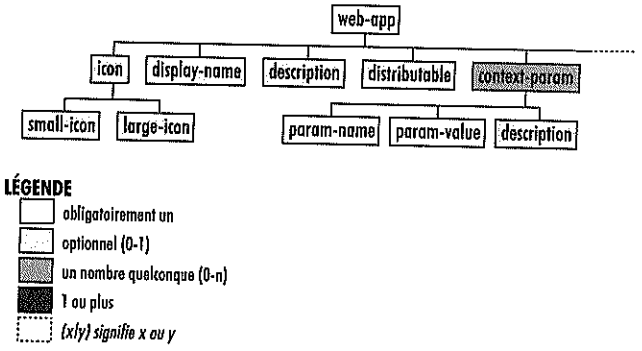


Figure C-1. Structure des éléments de la DTD du descripteur du déploiement

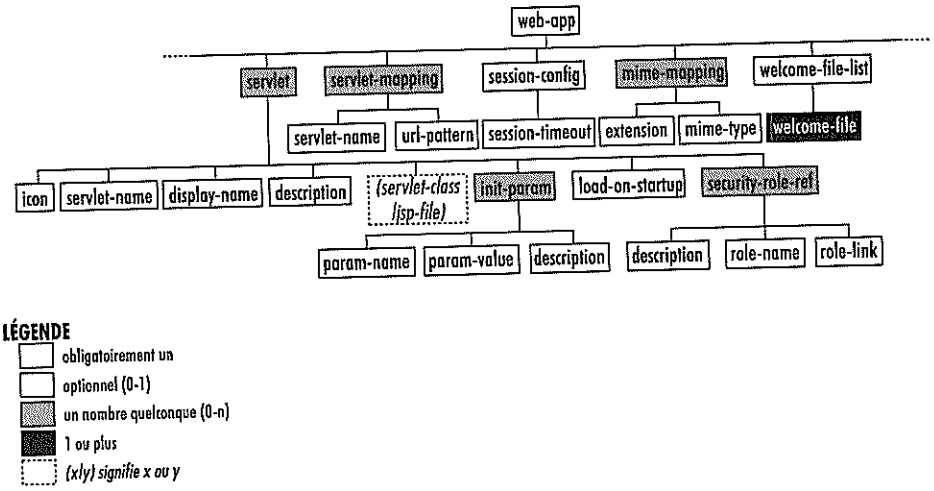


Figure C-2.

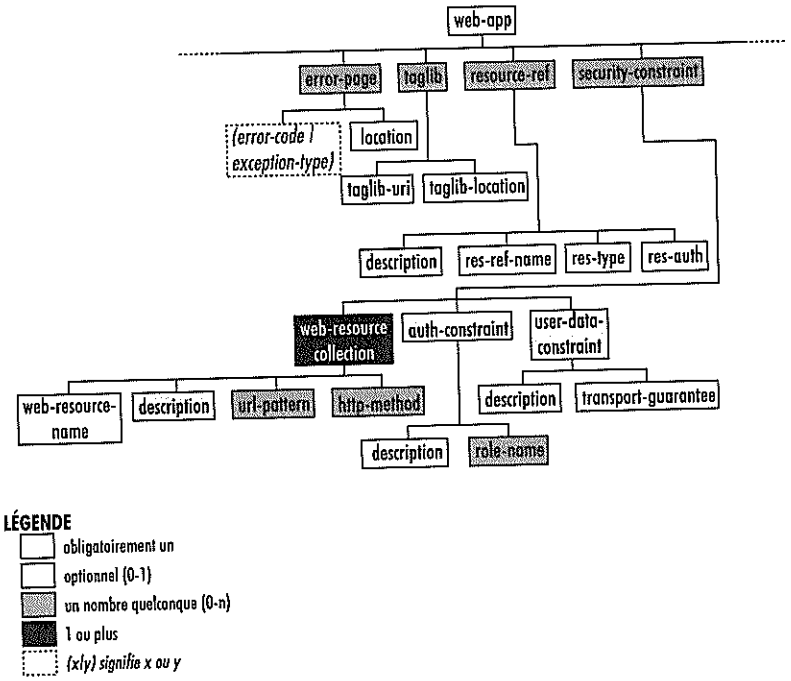


Figure C-3.

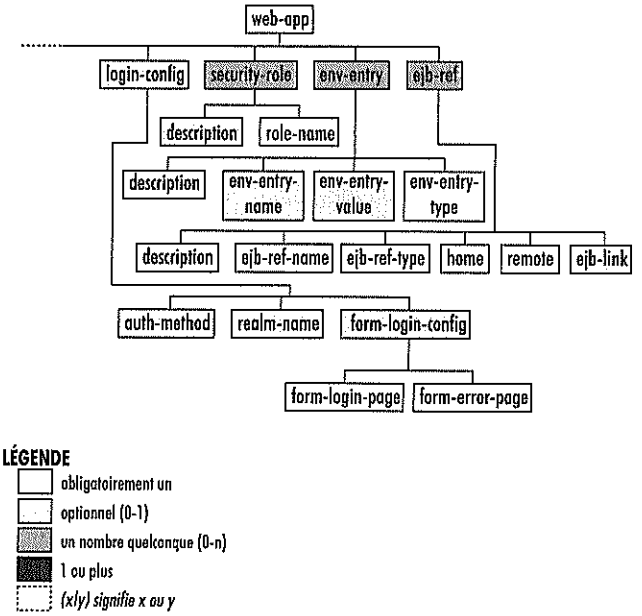


Figure C-4.

Exemple C-1. La DTD du descripteur de déploiement

```
<!ELEMENT web-app (icon?, display-name?, description?, distributable?,
                   context-param*, servlet*, servlet-mapping*, session-config?,
                   mime-mapping*, welcome-file-list?, error-page*, taglib*,
                   resource-ref*, security-constraint*, login-config?,
                   security-role*, env-entry*, ejb-ref*)>

<!ELEMENT icon (small-icon?, large-icon?)>
<!ELEMENT small-icon (#PCDATA)>
<!ELEMENT large-icon (#PCDATA)>
<!ELEMENT display-name (#PCDATA)>
<!ELEMENT description (#PCDATA)>

<!ELEMENT distributable EMPTY>

<!ELEMENT context-param (param-name, param-value, description?)>
<!ELEMENT param-name (#PCDATA)>
<!ELEMENT param-value (#PCDATA)>

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
                  (servlet-class|jsp-file), init-param*, load-on-startup?,
                   security-role-ref*)>
<!ELEMENT servlet-name (#PCDATA)>
<!ELEMENT servlet-class (#PCDATA)>
<!ELEMENT jsp-file (#PCDATA)>
<!ELEMENT init-param (param-name, param-value, description?)>
```

Exemple C-1. La DTD du descripteur de déploiement (suite)

```
<!ELEMENT load-on-startup (#PCDATA)>

<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
<!ELEMENT url-pattern (#PCDATA)>

<!ELEMENT session-config (session-timeout?)>
<!ELEMENT session-timeout (#PCDATA)>

<!ELEMENT mime-mapping (extension, mime-type)>
<!ELEMENT extension (#PCDATA)>
<!ELEMENT mime-type (#PCDATA)>

<!ELEMENT welcome-file-list (welcome-file+)>
<!ELEMENT welcome-file (#PCDATA)>

<!ELEMENT taglib (taglib-uri, taglib-location)>
<!ELEMENT taglib-uri (#PCDATA)>
<!ELEMENT taglib-location (#PCDATA)>

<!ELEMENT error-page ((error-code | exception-type), location)>
<!ELEMENT error-code (#PCDATA)>
<!ELEMENT exception-type (#PCDATA)>
<!ELEMENT location (#PCDATA)>

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT res-type (#PCDATA)>
<!ELEMENT res-auth (#PCDATA)>

<!ELEMENT security-constraint (web-resource-collection+,
                               auth-constraint?, user-data-constraint?)>
<!ELEMENT web-resource-collection (web-resource-name, description?,
                                   url-pattern*, http-method*)>
<!ELEMENT web-resource-name (#PCDATA)>
<!ELEMENT http-method (#PCDATA)>
<!ELEMENT user-data-constraint (description?, transport-guarantee)>
<!ELEMENT transport-guarantee (#PCDATA)>
<!ELEMENT auth-constraint (description?, role-name*)>
<!ELEMENT role-name (#PCDATA)>

<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>
<!ELEMENT realm-name (#PCDATA)>
<!ELEMENT form-login-config (form-login-page, form-error-page)>
<!ELEMENT form-login-page (#PCDATA)>
<!ELEMENT form-error-page (#PCDATA)>
<!ELEMENT auth-method (#PCDATA)>

<!ELEMENT security-role (description?, role-name)>
```

Exemple C-1. La DTD du descripteur de déploiement (suite)

```
<!ELEMENT security-role-ref (description?, role-name, role-link)>
<!ELEMENT role-link (#PCDATA)>

<!ELEMENT env-entry (description?, env-entry-name, env-entry-value?,
                     env-entry-type)>
<!ELEMENT env-entry-name (#PCDATA)>
<!ELEMENT env-entry-value (#PCDATA)>
<!ELEMENT env-entry-type (#PCDATA)>

<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote,
                   ejb-link?)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT ejb-ref-type (#PCDATA)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT remote (#PCDATA)>
<!ELEMENT ejb-link (#PCDATA)>

<!-- Le mécanisme des identifiants (ID) est destiné aux outils afin qu'ils
puissent faire facilement référence aux éléments du descripteur de déploiement.
Ils peuvent ainsi produire des informations de déploiement supplémentaires
(c'est-à-dire des informations en plus des informations standard données par le
descripteur de déploiement) permettant de stocker ces informations dans un
fichier séparé et de faire facilement référence à partir de ces fichiers aux
informations qui se trouvent dans le descripteur de déploiement standard d'une
application web. -->

<!ATTLIST web-app id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST distributable id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST servlet id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST jsp-file id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST url-pattern id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST mime-mapping id ID #IMPLIED>
<!ATTLIST extension id ID #IMPLIED>
```

Exemple C-1. La DTD du descripteur de déploiement (suite)

```
<!ATTLIST mime-type id ID #IMPLIED>
<!ATTLIST welcome-file-list id ID #IMPLIED>
<!ATTLIST welcome-file id ID #IMPLIED>
<!ATTLIST error-page id ID #IMPLIED>
<!ATTLIST error-code id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST location id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST web-resource-collection id ID #IMPLIED>
<!ATTLIST web-resource-name id ID #IMPLIED>
<!ATTLIST http-method id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
<!ATTLIST basic-auth id ID #IMPLIED>
<!ATTLIST form-auth id ID #IMPLIED>
<!ATTLIST form-login-page id ID #IMPLIED>
<!ATTLIST form-error-page id ID #IMPLIED>
<!ATTLIST form-login-config id ID #IMPLIED>
<!ATTLIST realm-name id ID #IMPLIED>
<!ATTLIST login-config id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST mutual-auth id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
```

<auth-constraint>

Synopsis

```
<!ELEMENT auth-constraint (description?, role-name*)>
```

Description

L'élément <auth-constraint> indique le rôle d'utilisateur qui est autorisé à accéder à un ensemble de ressources. Les noms de rôles doivent apparaître dans un élément <security-role-ref>.

```
<security-constraint>
    <web-resource-collection>
        ...
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
```

<auth-method>

Synopsis

```
<!ELEMENT auth-method (#PCDATA)>
```

Description

L'élément <auth-method> configure le mécanisme d'authentification pour l'application web. Pour obtenir accès à une ressource web protégée par une contrainte d'authentification, un utilisateur doit avoir été authentifié à l'aide du mécanisme configuré. Cet élément peut prendre les valeurs BASIC, DIGEST, FORM ou CLIENT-CERT.

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Default</realm-name>
</login-config>
```

<context-param>

Synopsis

```
<!ELEMENT context-param (param-name, param-value, description?)>
```

Description

L'élément <context-param> contient la déclaration des paramètres d'initialisation du contexte de servlet d'une application web.

```
<context-param>
    <param-name>rmihost</param-name>
    <param-value>localhost</param-value>
</context-param>
```

<description>

Synopsis

```
<!ELEMENT description (#PCDATA)>
```

Description

L'élément <description> fournit un texte de description de l'élément parent.

```
<description>Ne pas oublier de dire bonjour !</description>
```

<display-name>

Synopsis

```
<!ELEMENT display-name (#PCDATA)>
```

Description

L'élément <display-name> contient un nom court pour la servlet qui sera affiché par les outils graphiques.

```
<servlet>
    <servlet-name>hi</servlet-name>
    <display-name>HelloWorld</display-name>
    <servlet-class>HelloWorld</servlet-class>
</servlet>
```

<distributable>

Synopsis

```
<!ELEMENT distributable EMPTY>
```

Description

L'élément <distributable>, par sa présence dans un descripteur de déploiement, indique que l'application web est correctement programmée afin d'être déployée dans un conteneur de servlets distribué.

```
<web-app>
    <description>
        Toutes les servlets les pages JSP sont prêtes pour un déploiement
        distribué.
    </description>

    <distributable/>
...
```

<ejb-link>

Synopsis

```
<!ELEMENT ejb-link (#PCDATA)>
```

Description

L'élément <ejb-link> est utilisé dans l'élément <ejb-ref> pour spécifier qu'une référence d'EJB est lié à un EJB dans un paquetage d'application J2EE (*Java 2, Enterprise Edition*) englobant. La valeur de l'élément <ejb-link> doit être le <ejb-name> de l'EJB dans le paquetage d'application J2EE.

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
    <ejb-link>CabinBean</ejb-link>
</ejb-ref>
```

<ejb-ref>

Synopsis

```
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
                   remote, ejb-link?)>
```

Description

L'élément <ejb-ref> déclare une référence à un bean métier.

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

<ejb-ref-name>

Synopsis

```
<!ELEMENT ejb-ref-name (#PCDATA)>
```

Description

L'élément <ejb-ref-name> contient le nom d'une référence d'EJB. Il s'agit du nom JNDI que le code de la servlet utilise pour obtenir une référence au bean métier.

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

<ejb-ref-type>

Synopsis

```
<!ELEMENT ejb-ref-type (#PCDATA)>
```

Description

L'élément <ejb-ref-type> contient le type du bean. Sa valeur doit être Entity ou Session.[1]

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

<env-entry>

Synopsis

```
<!ELEMENT env-entry (description?, env-entry-name,
                     env-entry-value?, env-entry-type)>
```

Description

L'élément <env-entry> contient la déclaration d'une entrée de l'environnement de l'application, disponible *via* une recherche JNDI dans le contexte java:comp/env.

```
<env-entry>
    <description>Envoyer les codes PIN par courrier</description>
```

1. Le commentaire originale décrivant cette balise expliquait, de façon erronée, qu'elle « contient la classe Java attendue de l'EJB référencé ».

```
    <env-entry-name>mailPincode</env-entry-name>
    <env-entry-value>false</env-entry-value>
    <env-entry-type>java.lang.Boolean</env-entry-type> <!-- NCC -->
</env-entry>
```

<env-entry-name>

Synopsis

```
<!ELEMENT env-entry-name (#PCDATA)>
```

Description

L'élément <env-entry-name> spécifie le nom de recherche de l'entrée de l'environnement d'une application.

```
<env-entry>
    <description>Envoyer les codes PIN par courrier</description>
    <env-entry-name>mailPincode</env-entry-name>
    <env-entry-value>false</env-entry-value>
    <env-entry-type>java.lang.Boolean</env-entry-type> <!-- NCC -->
</env-entry>
```

<env-entry-type>

Synopsis

```
<!ELEMENT env-entry-type (#PCDATA)>
```

Description

L'élément <env-entry-type> contient le nom de type Java complet attendu par le code applicatif pour la valeur de l'entrée de l'environnement. Les valeurs suivantes sont valides : `java.lang.Boolean`, `java.lang.String`, `java.lang.Integer`, `java.lang.Double` et `java.lang.Float`.

```
<env-entry>
    <description>Envoyer les codes PIN par courrier</description>
    <env-entry-name>mailPincode</env-entry-name>
    <env-entry-value>false</env-entry-value>
    <env-entry-type>java.lang.Boolean</env-entry-type> <!-- NCC -->
</env-entry>
```

<env-entry-value>

Synopsis

```
<!ELEMENT env-entry-value (#PCDATA)>
```

Description

L'élément <env-entry-value> contient la valeur par défaut d'une entrée de l'environnement d'une application.

```
<env-entry>
    <description>Envoyer les codes PIN par courrier</description>
    <env-entry-name>mailPincode</env-entry-name>
    <env-entry-value>false</env-entry-value>
    <env-entry-type>java.lang.Boolean</env-entry-type> <!-- NCC -->
</env-entry>
```

<error-code>

Synopsis

```
<!ELEMENT error-code (#PCDATA)>
```

Description

L'élément <error-code> contient un code d'erreur HTTP.

```
<error-page>
    <error-code>400</error-code>
    <location>/400.html</location>
</error-page>
```

<error-page>

Synopsis

```
<!ELEMENT error-page ((error-code | exception-type), location)>
```

Description

L'élément <error-page> contient une correspondance entre un code d'erreur ou un type d'exception et le chemin d'une ressource dans une application web.

```
<error-page>
    <error-code>400</error-code>
    <location>/400.html</location>
</error-page>
```

<exception-type>

Synopsis

```
<!ELEMENT exception-type (#PCDATA)>
```

Description

L'élément <exception-type> contient un nom de classe complet d'un type d'exception Java.

```
<error-page>
    <exception-type>javax.servlet.ServletException</exception-type>
    <location>/servlet/ErrorDisplay</location>
</error-page>
```

<extension>

Synopsis

```
<!ELEMENT extension (#PCDATA)>
```

Description

L'élément <extension> contient une chaîne décrivant une extension de fichier.

```
<mime-mapping>
    <extension>wml</extension>
    <mime-type>text/vnd.wap.wml</mime-type>
</mime-mapping>
```

<form-error-page>

Synopsis

```
<!ELEMENT form-error-page (#PCDATA)>
```

Description

L'élément <form-error-page> définit l'emplacement dans l'application web de la page d'erreur affichée quand l'ouverture de session échoue.

```
<form-login-config>
    <form-login-page>/loginpage.html</form-login-page>
    <form-error-page>/errorpage.html</form-error-page>
</form-login-config>
```

<form-login-config>

Synopsis

```
<!ELEMENT form-login-config (form-login-page, form-error-page)>
```

Description

L'élément <form-login-config> spécifie la page d'ouverture de session et la page d'erreur devant servir lors d'une ouverture de session basée sur un formulaire. Si ce type d'authentification n'est pas utilisé, ces éléments sont ignorés.

```
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/loginpage.html</form-login-page>
        <form-error-page>/errorpage.html</form-error-page>
    </form-login-config>
</login-config>
```

<form-login-page>

Synopsis

```
<!ELEMENT form-login-page (#PCDATA)>
```

Description

L'élément <form-login-page> définit l'emplacement dans l'application web de la page d'ouverture de session.

```
<form-login-config>
    <form-login-page>/loginpage.html</form-login-page>
    <form-error-page>/errorpage.html</form-error-page>
</form-login-config>
```

<home>

Synopsis

```
<!ELEMENT home (#PCDATA)>
```

Description

L'élément <home> contient le nom complet de l'interface du Home de l'EJB.

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

<http-method>

Synopsis

```
<!ELEMENT http-method (#PCDATA)>
```

Description

L'élément <http-method> spécifie une méthode HTTP.

```
<web-resource-collection>
    <web-resource-name>SecretProtection</web-resource-name>
    <url-pattern>/servlet/SalaryServer</url-pattern>
    <url-pattern>/servlet/secret</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</web-resource-collection>
```

<icon>

Synopsis

```
<!ELEMENT icon (small-icon?, large-icon?)>
```

Description

L'élément <icon> peut contenir des éléments <small-icon> et <large-icon> spécifiant l'emplacement dans la structure de fichiers de l'application web des images, petite et grande, servant à représenter l'application web dans un outil graphique.

```
<icon>
    <small-icon>/images/little.gif</small-icon>
    <large-icon>/images/big.gif</large-icon>
</icon>
```

<init-param>

Synopsis

```
<!ELEMENT init-param (param-name, param-value, description?)>
```

Description

L'élément <init-param> contient une paire nom/valeur spécifiant un paramètre d'initialisation de la servlet.

```
<init-param>
    <param-name>key</param-name>
    <param-value>-9151314447111823249</param-value>
</init-param>
```

<jsp-file>

Synopsis

```
<!ELEMENT jsp-file (#PCDATA)>
```

Description

L'élément <jsp-file> contient le chemin d'un fichier JSP au sein de l'application web.

```
<servlet>
    <servlet-name>catalog-order</servlet-name>
    <servlet-class>Catalog</servlet-class>
    <jsp-file>/orderform.jsp</jsp-file>
</servlet>
```

<large-icon>

Synopsis

```
<!ELEMENT large-icon (#PCDATA)>
```

Description

L'élément <large-icon> spécifie l'emplacement au sein de l'application web d'un fichier contenant un grand icone (32×32 pixels). Les outils acceptent au moins des images au format GIF ou JPEG.

```
<icon>
    <small-icon>/images/little.gif</small-icon>
    <large-icon>/images/big.gif</large-icon>
</icon>
```

<load-on-startup>

Synopsis

```
<!ELEMENT load-on-startup (#PCDATA)>
```

Description

L'élément <load-on-startup> indique que la servlet doit être chargée au démarrage de l'application web. Le contenu optionnel de cet élément doit être un entier positif spécifiant l'ordre dans lequel la servlet doit être chargée. Les servlets ayant les valeurs les plus faibles sont chargées avant celles ayant des valeurs plus élevées. Si aucune valeur n'est donnée ou si elle n'est pas un entier positif, le conteneur est libre de charger la servlet à n'importe quel moment de la séquence de démarrage.

```
<servlet>
    <servlet-name>ps</servlet-name>
    <servlet-class>PrimeSearcher</servlet-class>
    <load-on-startup/>
</servlet>
```

<location>

Synopsis

```
<!ELEMENT location (#PCDATA)>
```

Description

L'élément <location> contient l'emplacement d'une ressource dans l'application web.

```
<error-page>
    <exception-type>javax.servlet.ServletException</exception-type>
    <location>/servlet/ErrorDisplay</location>
</error-page>
```

<login-config>

Synopsis

```
<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>
```

Description

L'élément <login-config> configure la méthode d'authentification et le nom de domaine utilisés pour l'application, ainsi que les attributs pour l'ouverture de session basée sur un formulaire, si elle est choisie.

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Default</realm-name>
</login-config>
```

<mime-mapping>

Synopsis

```
<!ELEMENT mime-mapping (extension, mime-type)>
```

Description

L'élément <mime-mapping> définit une correspondance entre une extension et un type MIME.

```
<mime-mapping>
    <extension>wml</extension>
    <mime-type>text/vnd.wap.wml</mime-type>
</mime-mapping>
```

<mime-type>

Synopsis

```
<!ELEMENT mime-type (#PCDATA)>
```

Description

L'élément <mime-type> contient la définition d'un type MIME.

```
<mime-mapping>
    <extension>wml</extension>
    <mime-type>text/vnd.wap.wml</mime-type>
</mime-mapping>
```

<param-name>

Synopsis

```
<!ELEMENT param-name (#PCDATA)>
```

Description

L'élément <param-name> contient le nom d'un paramètre.

```
<context-param>
    <param-name>rmihost</param-name>
    <param-value>localhost</param-value>
</context-param>
```

<param-value>

Synopsis

```
<!ELEMENT param-value (#PCDATA)>
```

Description

L'élément <param-value> contient la valeur d'un paramètre.

```
<init-param>
    <param-name>key</param-name>
    <param-value>-9151314447111823249</param-value>
</init-param>
```

<realm-name>

Synopsis

```
<!ELEMENT realm-name (#PCDATA)>
```

Description

L'élément <realm-name> spécifie le nom du domaine à utiliser dans l'authentification HTTP de base.

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Default</realm-name>
</login-config>
```

<remote>

Synopsis

```
<!ELEMENT remote (#PCDATA)>
```

Description

L'élément <remote> contient le nom complet de l'interface distant de l'EJB.

```
<ejb-ref>
    <description>Cabine d'un paquebot</description>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
</ejb-ref>
```

<res-auth>

Synopsis

```
<!ELEMENT res-auth (#PCDATA)>
```

Description

L'élément <res-auth> indique si le code du composant de l'application s'inscrit auprès d'une ressource par programmation ou si le conteneur s'inscrit auprès de la ressource d'après l'information de correspondance de Principal fournie par le déployeur. Doit prendre la valeur CONTAINER ou SERVLET.

```
<resource-ref>
    <description>Base de données principale</description>
    <res-ref-name>jdbc/primaryDB</res-ref-name>
```

```
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>CONTAINER</res-auth>
</resource-ref>
```

<res-ref-name>

Synopsis

```
<!ELEMENT res-ref-name (#PCDATA)>
```

Description

L'élément <res-ref-name> spécifie le nom d'une référence de fabrique de ressources.

```
<resource-ref>
    <description>Base de données principale</description>
    <res-ref-name>jdbc/primaryDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>CONTAINER</res-auth>
</resource-ref>
```

<res-type>

Synopsis

```
<!ELEMENT res-type (#PCDATA)>
```

Description

L'élément <res-type> spécifie le type (classe Java) de la source de données.

```
<resource-ref>
    <description>Base de données principale</description>
    <res-ref-name>jdbc/primaryDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>CONTAINER</res-auth>
</resource-ref>
```

<resource-ref>

Synopsis

```
<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>
```

Description

L'élément <resource-ref> contient une déclaration d'une référence de l'application web à une ressource externe, comme un composant EJB ou un DataSource JDBC.

```
<resource-ref>
    <description>Base de données principale</description>
    <res-ref-name>jdbc/primaryDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>CONTAINER</res-auth>
</resource-ref>
```

<role-link>

Synopsis

```
<!ELEMENT role-link (#PCDATA)>
```

Description

L'élément <role-link> relie une référence à un rôle de sécurité à un rôle de sécurité défini. L'élément doit contenir le nom d'un des rôles de sécurité défini dans les éléments <security-role>.

```
<security-role-ref>
    <role-name>mgr</role-name>
    <role-link>manager</role-link>
</security-role-ref>
```

<role-name>

Synopsis

```
<!ELEMENT role-name (#PCDATA)>
```

Description

L'élément <role-name> contient le nom d'un rôle de sécurité.

```
<auth-constraint>
    <role-name>manager</role-name>
</auth-constraint>
```

<security-constraint>

Synopsis

```
<!ELEMENT security-constraint (web-resource-collection+,
                               auth-constraint?,
                               user-data-constraint?)>
```

Description

L'élément <security-constraint> associe des contraintes de sécurité à un ou plusieurs ensembles de ressources web.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SecretProtection</web-resource-name>
        <url-pattern>/servlet/SalaryServer</url-pattern>
        <url-pattern>/servlet/secret</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
```

<security-role>

Synopsis

```
<!ELEMENT security-role (description?, role-name)>
```

Description

L'élément <security-role> contient la déclaration d'un rôle de sécurité utilisé dans les contraintes de sécurité de l'application web.

```
<security-role>
    <role-name>manager</role-name>
</security-role>
```

<security-role-ref>

Synopsis

```
<!ELEMENT security-role-ref (description?, role-name, role-link)>
```

Description

L'élément <security-role-ref> spécifie un alias pour un rôle de sécurité, c'est-à-dire une correspondance entre le nom utilisé par une servlet et celui utilisé dans le descripteur de déploiement.

```
<security-role-ref>
    <role-name>mgr</role-name>
    <role-link>manager</role-link>
</security-role-ref>
```

<servlet>

Synopsis

```
<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
                   (servlet-class|jsp-file), init-param*,
                   load-on-startup?, security-role-ref*)>
```

Description

L'élément <servlet> contient les données de déclaration d'une servlet, comme son nom d'enregistrement et son fichier de classe.

```
<servlet>
    <servlet-name>hi</servlet-name>
    <display-name>hi</display-name>
    <description>Ne pas oublier de dire bonjour !</description>
    <servlet-class>HelloWorld</servlet-class>
    <init-param>
        <param-name>defaultName</param-name>
        <param-value>World</param-value>
    </init-param>
    <load-on-startup/>
    <security-role-ref>
        <role-name>mgr</role-name>
        <role-link>manager</role-link>
    </security-role-ref>
</servlet>
```

<servlet-class>

Synopsis

```
<!ELEMENT servlet-class (#PCDATA)>
```

Description

L'élément <servlet-class> contient le nom de classe complet de la servlet.

```
<servlet>
    <servlet-name>hi</servlet-name>
    <servlet-class>HelloWorld</servlet-class>
</servlet>
```

<servlet-mapping>

Synopsis

```
<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

Description

L'élément <servlet-mapping> définit une correspondance entre une servlet et un motif d'URL.

```
<servlet-mapping>
    <servlet-name>hi</servlet-name>
    <url-pattern>/hello.html</url-pattern>
</servlet-mapping>
```

<servlet-name>

Synopsis

```
<!ELEMENT servlet-name (#PCDATA)>
```

Description

L'élément <servlet-name> spécifie le nom canonique de la servlet.

```
<servlet>
    <servlet-name>hi</servlet-name>
    <servlet-class>HelloWorld</servlet-class>
</servlet>
```

<session-config>

Synopsis

```
<!ELEMENT session-config (session-timeout?)>
```

Description

L'élément <session-config> définit les paramètres de session d'une application web.

```
<session-config>
    <session-timeout>60 <!-- minutes --></session-timeout>
</session-config>
```

<session-timeout>

Synopsis

```
<!ELEMENT session-timeout (#PCDATA)>
```

Description

L'élément <session-timeout> définit la durée d'expiration par défaut pour toutes les sessions crées dans l'application web. Elle doit être exprimée en un nombre entier de minutes.

```
<session-config>
    <session-timeout>60 <!-- minutes --></session-timeout>
</session-config>
```

<small-icon>

Synopsis

```
<!ELEMENT small-icon (#PCDATA)>
```

Description

L'élément <small-icon> spécifie l'emplacement au sein de l'application web d'un fichier contenant un petit icone (16×16 pixels). Les outils acceptent au moins des images au format GIF ou JPEG.

```
<icon>
    <small-icon>/images/little.gif</small-icon>
    <large-icon>/images/big.gif</large-icon>
</icon>
```

<taglib>

Synopsis

```
<!ELEMENT taglib (taglib-uri, taglib-location)>
```

Description

L'élément <taglib> décrit une bibliothèque de balises JSP.

```
<taglib>
    <taglib-uri>/WEB-INF/struts.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts.tld</taglib-location>
</taglib>
```

<taglib-location>

Synopsis

```
<!ELEMENT taglib-location (#PCDATA)>
```

Description

L'élément <taglib-location> contient l'emplacement (sous forme de ressource relative à la racine de l'application web) du fichier TLD (*Tag Library Description*) de la bibliothèque de balises.

```
<taglib>
    <taglib-uri>/WEB-INF/struts.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts.tld</taglib-location>
</taglib>
```

<taglib-uri>

Synopsis

```
<!ELEMENT taglib-uri (#PCDATA)>
```

Description

L'élément <taglib-uri> décrit un URI, relatif à l'emplacement du document *web.xml*, identifiant une bibliothèque de balises dans l'application web.

```
<taglib>
    <taglib-uri>/WEB-INF/struts.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts.tld</taglib-location>
</taglib>
```

<transport-guarantee>

Synopsis

```
<!ELEMENT transport-guarantee (#PCDATA)>
```

Description

L'élément <transport-guarantee> spécifie que les communications entre le client et le serveur doivent être NONE, INTEGRAL ou CONFIDENTIAL. NONE signifie que l'application n'a besoin d'aucune garantie sur le transport. La valeur INTEGRAL signifie que l'application a besoin que les données envoyées entre le client et le serveur le soit de manière à ne pouvoir être modifiées pendant le transit. CONFIDENTIAL signifie que l'application a besoin que ses données soient transmises de manière à interdire que d'autres entités observent le contenu de l'envoi. Dans la plupart des cas, la présence de INTEGRAL ou de CONFIDENTIAL indique l'emploi obligatoire de SSL.

```
<user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

<url-pattern>

Synopsis

```
<!ELEMENT url-pattern (#PCDATA)>
```

Description

L'élément <url-pattern> contient le motif d'URL d'une correspondance de servlet. Ce motif doit suivre les règles données dans la Section 10 de la spécification de l'API Servlet et expliquées au chapitre 2, *Introduction aux servlets HTTP*.

```
<servlet-mapping>
    <servlet-name>hi</servlet-name>
    <url-pattern>/hello.html</url-pattern>
</servlet-mapping>
```

<user-data-constraint>

Synopsis

```
<!ELEMENT user-data-constraint (description?, transport-guarantee)>
```

Description

L'élément <user-data-constraint> indique comment les données communiquées entre le client et le conteneur doivent être protégées.

```
<security-constraint>
    <web-resource-collection>
        ...
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

<web-app>

Synopsis

```
<!ELEMENT web-app (icon?, display-name?, description?,
                   distributable?, context-param*, servlet*,
                   servlet-mapping*, session-config?, mime-mapping*,
                   welcome-file-list?, error-page*, taglib*,
                   resource-ref*, security-constraint*,
                   login-config?, security-role*, env-entry*,
                   ejb-ref*)>
```

Description

L'élément <web-app> est l'élément racine du descripteur de déploiement d'une application web.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>hi</servlet-name>
        <servlet-class>HelloWorld</servlet-class>
    </servlet>
</web-app>
```

<web-resource-collection>

Synopsis

```
<!ELEMENT web-resource-collection (web-resource-name, description?,
                                   url-pattern*, http-method*)>
```

Description

L'élément <web-resource-collection> identifie, dans l'application web, un sous-ensemble de ressources et de méthodes HTTP sur ces ressources auxquelles s'applique une contrainte de sécurité. Si aucune méthode HTTP n'est indiquée, la contrainte de sécurité s'applique à toutes les méthodes HHTP.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SecretProtection</web-resource-name>
        <url-pattern>/servlet/SalaryServer</url-pattern>
        <url-pattern>/servlet/secret</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
</security-constraint>
```

<web-resource-name>

Synopsis

```
<!ELEMENT web-resource-name (#PCDATA)>
```

Description

L'élément <web-resource-name> spécifie le nom d'un ensemble de ressources web.

```
<web-resource-collection>
    <web-resource-name>SecretProtection</web-resource-name>
    <url-pattern>/servlet/SalaryServer</url-pattern>
    <url-pattern>/servlet/secret</url-pattern>
```

```
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</web-resource-collection>
```

<welcome-file>

Synopsis

```
<!ELEMENT welcome-file (#PCDATA)>
```

Description

L'élément <welcome-file> contient un nom de fichier utilisé comme page d'accueil par défaut.

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

<welcome-file-list>

Synopsis

```
<!ELEMENT welcome-file-list (welcome-file+)>
```

Description

L'élément <welcome-file-list> spécifie une liste de fichiers que le conteneur doit examiner quand un navigateur demande un répertoire au lieu d'une page web ou d'une servlet.

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

D

Codes d'état HTTP

Les codes d'état HTTP sont regroupés comme le montre le *tableau D-1*.

Tableau D-1. Regroupement des codes d'état HTTP

Intervalle	Signification de la réponse
100-199	Information
200-299	Requête du client réussie
300-399	Requête du client redirigée, une autre action est nécessaire
400-499	Requête du client incomplète
500-599	Erreur du serveur

Le *tableau D-2* liste les constantes des codes d'état HTTP définies dans l'interface `HttpServletResponse` et utilisées comme paramètres aux méthodes `setStatus()` et `sendError()`. Le numéro de version dans la dernière colonne fait référence à la version du protocole HTTP qui a défini en premier le code d'état. L'API Servlet 2.0 a ajouté des constantes pour les codes d'état de HTTP Version 1.1 comme il est spécifié par la norme proposée dans la RFC 2068 ; l'API Servlet 2.2 a ajouté les codes d'état 416 et 417 comme il est spécifié par le brouillon de norme dans la RFC 2616. Notez que les codes d'état de HTTP/1.1 demandent un navigateur compatible HTTP/1.1.

Pour plus d'informations sur HTTP, reportez-vous au livre *Web Client Programming* de Clinton Wong (O'Reilly & Associates). La dernière spécification HTTP/1.1 est disponible dans la RFC 2616 à *http://www.ietf.org/rfc/rfc2616.txt*.

Tableau D-2. Constantes des codes d'état HTTP

Constante	Code	Message par défaut	Signification	Version HTTP
SC_CONTINUE	100	Continue	Le serveur a reçu la partie initiale de la requête et le client peut continuer avec le reste de la requête.	1.1
SC_SWITCHING_PROTOCOLS	101	Switching Protocols	Le serveur consent à se soumettre à la demande du client de changer les protocoles par ceux spécifiés dans l'en-tête Upgrade de la requête. Cela peut inclure le passage à une nouvelle version de HTTP ou à un protocole personnalisé de synchronisation de flux vidéo.	1.1
SC_OK	200	OK	La requête du client a réussi et la réponse du serveur contient les données requises. C'est le code d'état par défaut.	1.0
SC_CREATED	201	Created	Une ressource a été créée sur le serveur, sans doute en réponse à une requête cliente. Le corps de la réponse inclut la ou les URL où l'on peut trouver la nouvelle ressource, la plus spécifique se trouvant dans l'en-tête Location. Si la ressource ne peut être créée immédiatement, un code d'état SC_ACCEPTED doit être retourné à la place.	1.0
SC_ACCEPTED	202	Accepted	La requête a été acceptée pour traitement, mais elle n'est pas encore terminée. Le serveur doit décrire l'état courant de la requête dans le corps de la réponse. Le serveur n'est pas obligé d'agir ou de terminer la requête.	1.0
SC_NON_AUTHORITATIVE_INFORMATION	203	Non-Authoritative Information	Les en-têtes HTTP de la réponse viennent d'une source locale ou tierce, au lieu du serveur original. Les servlets normales n'ont pas de raison d'utiliser ce code d'état.	1.1

Tableau D-2. Constantes des codes d'état HTTP (suite)

Constante	Code	Message par défaut	Signification	Version HTTP
SC_NO_CONTENT	204	No Content	La requête a réussi mais il n'y a pas de nouveau corps de réponse à retourner. Les navigateurs recevant ce code doivent garder leur vue courante du document. C'est un code utile quand une servlet accepte des données d'un formulaire mais veut que l'affichage du navigateur reste sur le formulaire, puisqu'il évite le message d'erreur « Document contains no data ».	1.0
SC_RESET_CONTENT	205	Reset Content	La requête a réussi et le navigateur doit remettre à zéro (recharger) la vue courante du document. C'est un code utile quand une servlet accepte des données d'un formulaire et veut que le formulaire rafraîchisse son état.	1.1
SC_PARTIAL_CONTENT	206	Partial Content	Le serveur a terminé une requête GET partielle et a retourné la portion du document indiquée dans l'en-tête Range du client.	1.1
SC_MULTIPLE_CHOICES	300	Multiple Choices	L'URL demandée fait référence à plus d'une ressource. Par exemple, l'URL peut faire référence à un document traduit en plusieurs langues. Le corps de la réponse doit expliquer les options du client dans un format approprié au type de contenu de la réponse. Le serveur peut suggérer un choix par l'en-tête Location.	1.1
SC_MOVED_PERMANENTLY	301	Moved Permanently	La ressource demandée a été déplacée de façon permanente vers un nouvel emplacement. Les références futures doivent utiliser la nouvelle URL dans leurs requêtes. Le nouvel emplacement est donné par l'en-tête Location. La plupart des navigateurs accèdent automatiquement au nouvel emplacement.	1.0

Tableau D-2. Constantes des codes d'état HTTP (suite)

Constante	Code	Message par défaut	Signification	Version HTTP
SC_MOVED_TEMPORARILY	302	Moved Temporarily	La ressource demandée a été déplacée de façon temporaire vers un nouvel emplacement, mais les références futures doivent continuer à utiliser l'URL originale pour accéder à la ressource. Le nouvel emplacement est donné dans l'en-tête Location. La plupart des navigateurs accèdent automatiquement au nouvel emplacement.	1.0
SC_SEE_OTHER	303	See Other	La ressource demandée a traité la requête mais le client doit obtenir sa réponse en effectuant un GET sur l'URL spécifiée dans l'en-tête Location. Ce code est utile quand une servlet veut recevoir les données POST puis rediriger le client vers une autre ressource pour la réponse.	1.1
SC_NOT_MODIFIED	304	Not Modified	Le document demandé n'a pas changé depuis la date spécifiée dans l'en-tête If-Modified-Since de la requête. Les servlets normales ne doivent pas utiliser ce code d'état. À la place, elles implémentent getLastModified().	1.0
SC_USE_PROXY	305	Use Proxy	La ressource demandée doit être accédée *via* le proxy donné dans l'en-tête Location.	1.1
SC_BAD_REQUEST	400	Bad Request	Le serveur ne peut comprendre la requête, probablement à cause d'une erreur de syntaxe.	1.0
SC_UNAUTHORIZED	401	Unauthorized	La requête n'a pas les bonnes autorisations. Utilisé avec les en-têtes WWW-Authenticate et Authorization.	1.0
SC_PAYMENT_REQUIRED	402	Payment Required	Réservé pour une utilisation future. Des propositions existent pour utiliser ce code avec l'en-tête Charge-To, mais ce n'est pas encore normalisé au moment de l'écriture de ce livre.	1.1

Tableau D-2. Constantes des codes d'état HTTP (suite)

Constante	Code	Message par défaut	Signification	Version HTTP
SC_FORBIDDEN	403	Forbidden	La requête a été comprise, mais le serveur ne va pas la réaliser. Le serveur peut en expliquer la raison dans le corps de la réponse.	1.0
SC_NOT_FOUND	404	Not Found	La ressource demandée n'a pas été trouvée ou n'est pas disponible.	1.0
SC_METHOD_NOT_ALLOWED	405	Method Not Allowed	La méthode utilisée par le client n'est pas supportée par cette URL. La liste des méthodes supportées doit être donnée dans l'en-tête Allow de la réponse.	1.1
SC_NOT_ACCEPTABLE	406	Not Acceptable	La ressource demandée existe, mais n'est pas dans un format acceptable pour le client (comme indiqué dans le ou les en-têtes Accept dans la requête).	1.1
SC_PROXY_AUTHENTICATION_REQUIRED	407	Proxy Authentication Required	Le serveur mandataire a besoin d'une autorisation avant de continuer. Utilisé avec l'en-tête Proxy-Authenticate. Les servlets normales ne doivent pas avoir besoin d'utiliser ce code d'état.	1.1
SC_REQUEST_TIMEOUT	408	Request Timeout	Le client n'a pas complètement fini sa requête pendant le temps d'écoute consenti par le serveur.	1.1
SC_CONFLICT	409	Conflict	La requête ne pourra pas être terminée car elle entre en conflit avec une autre requête ou la configuration du serveur. Ce code a le plus de chances d'arriver avec les requêtes PUT HTTP, où le fichier en train d'être déposé est sous un contrôle de révision et la nouvelle version entre en conflit avec des modifications précédentes. Le serveur peut envoyer une description du conflit dans le corps de la réponse.	1.0
SC_GONE	410	Gone	La ressource n'est plus disponible sur ce serveur et aucune autre adresse n'est connue. Ce code doit être utilisé uniquement lorsque la ressource a été déplacée de façon permanente. Les servlets n'ont pas de raison d'utiliser ce code d'état.	1.1

Tableau D-2. Constantes des codes d'état HTTP (suite)

Constante	Code	Message par défaut	Signification	Version HTTP
SC_LENGTH_REQUIRED	411	Length Required	Le serveur n'acceptera pas les requêtes sans un en-tête Content-Length.	1.1
SC_PRECONDITION_FAILED	412	Precondition Failed	Une précondition spécifiée par un ou plusieurs en-têtes If... dans la requête vaut false.	1.1
SC_REQUEST_ENTITY_TOO_LARGE	413	Request Entity Too Large	Le serveur ne traitera pas la requête car son contenu est trop grand. Si cette limitation est temporaire, le serveur peut inclure un en-tête Retry-After.	1.1
SC_REQUEST_URI_TOO_LONG	414	Request-URI Too Long	Le serveur ne traitera pas la requête parce que l'URI de la requête est plus long que ce qui est autorisé par le serveur. Cela peut arriver lorsqu'un client a converti accidentellement une requête POST en une requête GET. Les servlets normales n'ont aucune raison d'utiliser ce code d'état.	1.1
SC_UNSUPPORTED_MEDIA_TYPE	415	Unsupported Media Type	Le serveur ne traitera pas la requête car le corps de la requête a un format non supporté par la ressource demandée.	1.1
SC_REQUESTED_RANGE_NOT_SATISFYABLE	416	Requested Range Not Satisfiable	L'intervalle du contenu demandé par le client *via* l'en-tête Range ne correspond pas à la taille de la ressource demandée.	1.1 (RFC 2616)
SC_EXPECTATION_FAILED	417	Expectation Failed	La demande effectuée par le client via l'en-tête Expect ne peut être satisfaite.	1.1 (RFC 2616)
SC_INTERNAL_SERVER_ERROR	500	Internal Server Error	Une erreur inattendue est arrivée dans le serveur l'empêchant de terminer la requête.	1.0
SC_NOT_IMPLEMENTED	501	Not Implemented	Le serveur ne supporte pas la fonctionnalité nécessaire pour compléter la requête.	1.0
SC_BAD_GATEWAY	502	Bad Gateway	Un serveur agissant comme passerelle ou proxy n'a pas reçu une réponse valide d'un serveur en amont.	1.0

Tableau D-2. Constantes des codes d'état HTTP (suite)

Constante	Code	Message par défaut	Signification	Version HTTP
SC_SERVICE_UNAVAILABLE	503	Service Unavailable	Le service (serveur) est temporairement indisponible mais devrait être restauré dans le futur. Si le serveur sait quand il sera à nouveau disponible, un en-tête Retry-After peut aussi être fourni.	1.0
SC_GATEWAY_TIMEOUT	504	Gateway Timeout	Un serveur agissant comme passerelle ou proxy n'a pu recevoir une réponse valide d'un serveur en amont pendant le temps qu'il était prêt à attendre.	1.1
SC_HTTP_VERSION_NOT_SUPPORTED	505	HTTP Version Not Supported	Le serveur n'accepte pas la version du protocole HTTP utilisé dans la requête. Le corps de la réponse devrait spécifier les protocoles supportés par le serveur. Les servlets normales ne doivent pas avoir besoin d'utiliser ce code d'état.	1.1

E

Entités caractères

Le *tableau E-1* donne la liste des échappements Unicode, des entités HTML numériques et des entités HTML nommées pour tous les jeux de caractères ISO-8859-1 (Latin-1) imprimables.

Les entités numériques et nommées peuvent être utilisées dans les pages HTML ; elles sont converties en symboles par les navigateurs web. Les échappements Unicode peuvent être utilisés dans le code d'une servlet ; ils sont interprétés par le compilateur Java. Par exemple, le signe Livre (£) peut être embarqué dans une page HTML sous la forme `"£"` ou `"£"`. Il peut être mis directement dans du code Java sous la forme `"\u00A3"`.

Toutes les entités caractères HTML ne sont pas supportées universellement. La colonne Support indique leur niveau d'acceptation. Un *S* signifie que les entités numériques et nommées du symbole font partie de la norme HTML. Un *P* indique que les valeurs sont des normes proposées, elles ne font pas partie de la norme HTML mais elles sont généralement acceptées. Un *N* dans la colonne indique que les entités ne sont pas standards et peu acceptées. Pour ces symboles, il est préférable d'utiliser les échappements Unicode.

Tableau E-1. Entités caractères

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u0009			\t	Tabulation horizontale	S
\u000A
	\n	Saut de ligne	S
\u000D
	\r	Retour chariot	S
\u0020				Espace	S
\u0021	!		!	Point d'exclamation	S
\u0022	"	"	"	Guillemet	S
\u0023	#		#	Dièse	S
\u0024	$		$	Dollar	S

Tableau E-1. Entités caractères (suite)

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u0025	%		%	Pourcentage	S
\u0026	&	&	&	Esperluette	S
\u0027	'		'	Apostrophe	S
\u0028	(		(	Parenthèse ouvrante	S
\u0029	)		)	Parenthèse fermante	S
\u002A	*		*	Astérisque	S
\u002B	+		+	Plus	S
\u002C	,		,	Virgule	S
\u002D	-		-	Tiret	S
\u002E	.		.	Point	S
\u002F	/		/	Barre oblique	S
\u0030-\u0039	0-9		0–9	Chiffre 0–9	S
\u003A	:		:	Deux-points	S
\u003B	;		;	Point-virgule	S
\u003C	<	<	<	Inférieur à	S
\u003D	=		=	Égal	S
\u003E	>	>	>	Supérieur à	S
\u003F	?		?	Point d'interrogation	S
\u0040	@		@	Arobase	S
\u0041-\u005A	A-Z		A–Z	Lettres A–Z	S
\u005B	[		[	Crochet ouvrant	S
\u005C	\		\	Barre oblique inverse	S
\u005D	]		]	Crochet fermant	S
\u005E	^		^	Circonflexe	S
\u005F	_		_	Souligné	S
\u0060	`		`	Accent grave	S
\u0061-\u007A	a-z		a–z	Lettres a–z	S
\u007B	{		{	Accolade ouvrante	S
\u007C				|	Barre verticale	S
\u007D	}		}	Accolade fermante	S
\u007E	~		~	Tilde	S
\u0082	‚		‚	Apostrophe basse	N
\u0083	ƒ		*f*	Florin	N
\u0084	„		„	Guillemet droit bas	N

Tableau E-1. Entités caractères (suite)

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u0085	…		…	Points de suspension	N
\u0086	†		†	Dague	N
\u0087	‡		‡	Double dague	N
\u0088	ˆ		ˆ	Accent circonflexe	N
\u0089	‰		‰	Pour mille	N
\u008A	Š		Š	S majuscule, *caron*	N
\u008B	‹		<	Inférieur à	N
\u008C	Œ		Œ	OE majuscule	N
\u0091	‘		‘	Apostrophe gauche	N
\u0092	’		’	Apostrophe droite	N
\u0093	“		“	Guillemet gauche	N
\u0094	”		”	Guillemet droit	N
\u0095	•		•	Puce	N
\u0096	–		–	Tiret En	N
\u0097	—		—	Tiret Em	N
\u0098	˜		˜	Tilde	N
\u0099	™		™	Trademark	N
\u009A	š		š	s minscule, *caron*	N
\u009B	›		>	Supérieur à	N
\u009C	œ		œ	oe minuscule	N
\u009F	Ÿ		Ÿ	Y majuscule, tréma	N
\u00A0				Espace insécable	P
\u00A1	¡	¡	¡	Point d’exclamation inversé	P
\u00A2	¢	¢	¢	Cent	P
\u00A3	£	£	£	Livre	P
\u00A4	¤	¤		Monnaie générale	P
\u00A5	¥	¥	¥	Yen	P
\u00A6	¦	¦	|	Barre verticale coupée	P
\u00A7	§	§	§	Section	P
\u00A8	¨	¨	¨	Tréma	P
\u00A9	©	©	©	Copyright	P
\u00AA	ª	ª	ª	Ordinal féminin	P
\u00AB	«	«	«	Chevron ouvrant	P
\u00AC	¬	¬	¬	Non	P
\u00AD			-	Tiret doux	P
\u00AE	®	®	®	Registered trademark	P

Tableau E-1. Entités caractères (suite)

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u00AF	¯	¯	¯	Macron	P
\u00B0	°	°	°	Degré	P
\u00B1	±	±	±	Plus ou moins	P
\u00B2	²	²	²	2 exposant	P
\u00B3	³	³	³	3 exposant	P
\u00B4	´	´	´	Accent aigu	P
\u00B5	µ	µ	µ	Micro (mu grec)	P
\u00B6	¶	¶	¶	Paragraphe	P
\u00B7	·	·	·	Point milieu	P
\u00B8	¸	¸	¸	Cédille	P
\u00B9	¹	¹	¹	1 exposant	P
\u00BA	º	º	º	Ordinal masculin	P
\u00BB	»	»	»	Chevron fermant	P
\u00BC	¼	¼	¼	Un-quart	P
\u00BD	½	½	½	Un-demi	P
\u00BE	¾	¾	¾	Trois-quart	P
\u00BF	¿	¿	¿	Point d'interrogation inversé	P
\u00C0	À	À	À	A accent grave majuscule	S
\u00C1	Á	Á	Á	A accent aigu majuscule	S
\u00C2	Â	Â	Â	A accent circonflexe majuscule	S
\u00C3	Ã	Ã	Ã	A tilde majuscule	S
\u00C4	Ä	Ä	Ä	A tréma majuscule	S
\u00C5	Å	Å	Å	A cercle majuscule	S
\u00C6	Æ	Æ	Æ	AE majuscule	S
\u00C7	Ç	Ç	Ç	C cédille majuscule	S
\u00C8	È	È	È	E accent grave majuscule	S
\u00C9	É	É	É	E accent aigu majuscule	S
\u00CA	Ê	Ê	Ê	E accent circonflexe majuscule	S
\u00CB	Ë	Ë	Ë	E tréma majuscule	S
\u00CC	Ì	Ì	Ì	I accent grave majuscule	S
\u00CD	Í	Í	Í	I accent aigu majuscule	S
\u00CE	Î	Î	Î	I accent circonflexe majuscule	S
\u00CF	Ï	Ï	Ï	I tréma majuscule	S
\u00D0	Ð	Ð	Ð	Eth majuscule	S
\u00D1	Ñ	Ñ	Ñ	N tilde majuscule	S
\u00D2	Ò	Ò	Ò	O accent grave majuscule	S

Tableau E-1. Entités caractères (suite)

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u00D3	Ó	Ó	Ó	O accent aigu majuscule	S
\u00D4	Ô	Ô	Ô	O accent circonflexe majuscule	S
\u00D5	Õ	Õ	Õ	O tilde majuscule	S
\u00D6	Ö	Ö	Ö	O tréma majuscule	S
\u00D7	×	×	×	Multiplication	P
\u00D8	Ø	Ø	Ø	O majuscule, barré	S
\u00D9	Ù	Ù	Ù	U accent grave majuscule	S
\u00DA	Ú	Ú	Ú	U accent aigu majuscule	S
\u00DB	Û	Û	Û	U accent circonflexe majuscule	S
\u00DC	Ü	Ü	Ü	U tréma majuscule	S
\u00DD	Ý	Ý	Ý	Y accent aigu majuscule	S
\u00DE	Þ	Þ	Þ	Thorn majuscule	S
\u00DF	ß	ß	ß	sz minuscule, ligature	S
\u00E0	à	à	à	a accent grave minuscule	S
\u00E1	á	á	á	a accent aigu minuscule	S
\u00E2	â	â	â	a accent circonflexe minuscule	S
\u00E3	ã	ã	ã	a tilde minuscule	S
\u00E4	ä	ä	ä	a tréma minuscule	S
\u00E5	å	å	å	a cercle minuscule	S
\u00E6	æ	æ	æ	ae minuscule	S
\u00E7	ç	ç	ç	c cédille minuscule	S
\u00E8	è	è	è	e accent grave minuscule	S
\u00E9	é	é	é	e accent aigu minuscule	S
\u00EA	ê	ê	ê	e accent circonflexe minuscule	S
\u00EB	ë	ë	ë	e tréma minuscule	S
\u00EC	ì	ì	ì	i accent grave minuscule	S
\u00ED	í	í	í	i accent aigu minuscule	S
\u00EE	î	î	î	i accent circonflexe minuscule	S
\u00EF	ï	ï	ï	i tréma minuscule	S
\u00F0	ð	ð	ð	eth minuscule	S
\u00F1	ñ	ñ	ñ	n tilde miniscule	S
\u00F2	ò	ò	ò	o accent grave minuscule	S
\u00F3	ó	ó	ó	o accent aigu minuscule	S
\u00F4	ô	ô	ô	o accent circonflexe minuscule	S
\u00F5	õ	õ	õ	o tilde minuscule	S
\u00F6	ö	ö	ö	o tréma minuscule	S

Tableau E-1. Entités caractères (suite)

Échap. Unicode	Entité numérique	Entité nommée	Symbole	Description	Support
\u00F7	÷	÷	÷	Division	P
\u00F8	ø	ø	ø	o barré minuscule	S
\u00F9	ù	ù	ù	u accent grave minuscule	S
\u00FA	ú	ú	ú	u accent aigu minuscule	S
\u00FB	û	û	û	u accent circonflexe minuscule	S
\u00FC	ü	ü	ü	u tréma minuscule	S
\u00FD	ý	ý	ý	y accent aigu minuscule	S
\u00FE	þ	þ	þ	thorn minuscule	S
\u00FF	ÿ	ÿ	ÿ	y tréma minuscule	S

F

Charsets

Le *tableau F-1* donne la liste des charsets suggérés pour certaines langues. Les charsets sont utilisés par les servlets pour générer une sortie multilingue ; ils déterminent le codage de caractère utilisé par le `PrintWriter` d'une servlet. Par défaut, le `PrintWriter` utilise le charset ISO-8859-1 (Latin-1), approprié pour la plupart des langues occidentales. Pour spécifier un autre charset, sa valeur doit être passée à la méthode `setContentType()` avant que la servlet ne retrouve son `PrintWriter`. Par exemple :

```
res.setContentType("text/html; charset=Shift_JIS");// Un charset Japonais.
PrintWriter out = res.getWriter();    // Écrit en Japonais Shift_JIS
```

Le charset peut aussi être défini implicitement à l'aide de la méthode `setLocale()` :

```
res.setContentType("text/html");
res.setLocale(new Locale("ja", ""));  // Fixe le charset à Shift_JIS.
PrintWriter out = res.getWriter();    // Écrit en Japonais Shift_JIS.
```

La méthode `setLocale()` attribue un charset à la réponse en fonction de la table donnée ici. Quand plusieurs charsets sont possibles, le premier de la liste est choisi.

Notez que tous les serveurs web ne supportent pas tous les charsets ou disposent des polices pour représenter tous les caractères, bien que tous les clients acceptent au minimum le ISO-8859-1. Le charset UTF-8 peut représenter tous les caractères Unicode et on peut supposer qu'il est une solution viable pour toutes les langues.

Tableau F-1. Jeux de caractères suggérés

Langue	Code de langue	Charset suggéré
Albanais	sq	ISO-8859-2
Allemand	de	ISO-8859-1
Anglais	en	ISO-8859-1
Arabe	ar	ISO-8859-6
Bulgare	bg	ISO-8859-5
Biélorusse	be	ISO-8859-5
Catalan (Espagnol)	ca	ISO-8859-1

Tableau F-1. Jeux de caractères suggérés (suite)

Langue	Code de langue	Charset suggéré
Chinois (Simplifié/Continental)	zh	GB2312
Chinois (Traditionnel/Taiwan)	zh (pays TW)	Big5
Coréen	ko	EUC-KR[a]
Croate	hr	ISO-8859-2
Danois	da	ISO-8859-1
Espagnol	es	ISO-8859-1
Estonien	et	ISO-8859-1
Finnois	fi	ISO-8859-1
Français	fr	ISO-8859-1
Grec	el	ISO-8859-7
Hollandais	nl	ISO-8859-1
Hongrois	hu	ISO-8859-2
Hébreux	he (aussi iw)	ISO-8859-8
Islandais	is	ISO-8859-1
Italien	it	ISO-8859-1
Japonais	ja	Shift_JIS, ISO-2022-JP, EUC-JP[b]
Lettonien, Letton	lv	ISO-8859-2
Lituanien	lt	ISO-8859-2
Macédonien	mk	ISO-8859-5
Norvégien	no	ISO-8859-1
Polonais	pl	ISO-8859-2
Portugais	pt	ISO-8859-1
Roumain	ro	ISO-8859-2
Russe	ru	ISO-8859-5, KOI8-R
Serbe	sr	ISO-8859-5, KOI8-R
Serbo-Croate	sh	ISO-8859-5, ISO-8859-2, KOI8-R
Slovaque	sk	ISO-8859-2
Slovène	sl	ISO-8859-2
Suédois	sv	ISO-8859-1
Tchèque	cs	ISO-8859-2
Turc	tr	ISO-8859-9
Ukrainien	uk	ISO-8859-5, KOI8-R

a. Supporté pour la première fois dans le JDK 1.1.6. Les versions précédentes du JDK connaissent le jeu de caractères EUC-KR sous le nom KSC_5601. Pour des raisons de portabilité, vous pouvez donc fixer le charset à EUC-KR et construire manuellement un `PrintWriter` KSC_5601.

b. Supporté pour la première fois dans le JDK 1.1.6. Les versions précédentes du JDK connaissent le charset EUC-JP sous le nom EUCJIS. Pour des raisons de portabilité, vous pouvez donc fixer le charset à EUC-JP et construire manuellement un `PrintWriter` EUCJIS.

Index

Symboles

!= (comparaison), opérateur 439
<% ... %>, commande, insertion de contenu avec 427
<%– –%>, inclusion de commentaire JSP 494
<% %>, inclusion de scriptlets 490
<%! %>, inclusion de déclarations JSP 494
<%= %>, inclusion d'expressions JSP 494
<%@ %>, inclusion de directives JSP 495
&&, || et !, opérateurs booléens 439
/servlet/, préfixe, utilisation 20
== (comparaison), opérateur 439
{ } (parenthèses), délimitation des blocs de code avec 439

A

<A HREF>, liens (HTML), trouver et lister 526–529
accept() 313, 460
Accept, en-tête, valeurs multiples pour 109
Accept-Charset, en-tête 385
  HelloBabel (exemple), utilisation dans 386
Accept-Encoding, en-tête 183, 185
Accept-Language, en-tête 109, 385, 393
  HelloBabel (exemple), utilisation dans 386
accès
  autorisation basée sur les rôles 221
  aux servlets, restriction 222–225
  contrôle avec les gestionnaires de sécurité 12
  restriction des utilisateurs 84–85
  restriction pour les clients 83
accolades, { et }, démarcation de blocs de code pour les directives WebMacro 439
accueil personnalisé des clients 86
Acme GIF, codeur utilisant la compression LZ 169
ACTION, paramètre (formulaires), information de chemin supplémentaire dans 92
actions personnalisées (voir bibliothèques de balises personnalisées)
actions, JSP 500
  contraintes pour être du XML bien formé 502
  <jsp:forward> 506
  <jsp:include> 505
  <jsp:setProperty> 502
    page toolview.jsp, utilisation dans 509
  <jsp:useBean>
    attributs, définition 500
    corps, interprétation 501
    page toolview.jsp, utilisation dans 509
    toolview-tag.jsp (exemple), utilisation dans 515
  personnalisées 512–516
activation (session) 553
Active Server Pages (ASP) 5
addCookie() (HttpServletResponse) 199
addNotify() 168
adresse IP 191
  machine du client, obtention 82
  serveur, verrouillage d'une servlet à 76
adresses (émetteur et destinataire), messages e-mail 525
âge (maximum), spécification pour les cookies 200
algorithmes de chiffrement
  DES (Data Encryption Standard) 238
  integral et confidential, qualifications 242

secure, qualification 243
test de la taille 558
algorithmes de compression, fichiers d'image 169
alias pour les rôles
API Servlet 2.3, changements dans 561
création dans le descripteur de déploiement 225
Allaire
JRun Server 9
conteneur de servlets additionnel 9
ServletDebugger 543
analyse des versions de la servlet et de Java 82
analyse syntaxique
analyseur SAX 420
analyseur Tidy, conversion du HTML en DOM 470
modèles WebMacro 446
multipart/form-data 112–121
paramètres de requête 517–522
classe ParameterParser 518–522
ParameterNotFoundException 522
texte d'un modèle WebMacro avec un autre analyseur 441
animations, mise à jour par push du serveur 186
anonyme, suivi de session 193
champs de formulaire cachés, utilisation 193–195
réécriture d'URL, utilisation 196–198
anonymous, nom d'utilisateur 233
ANSI SQL-2 Entry Level 250
Apache
bibliothèque Village 286
classes de l'API Servlet, rendre disponibles 6
JServ et Tomcat, liaisons de sessions à 364
modèle de licence
Tea, utilisation du 399
WebMacro, utilisation du 429
module JServ (conteneur de servlets additionnel) 9
paquetages ECS 456
Tomcat Server ix, 7
comme conteneur additionnel 9
comme conteneur embarquable 9
mod_perl, utilisation avec 3
racine_serveur/webapps/ROOT, correspondance de contexte par défaut 26
servlets, gestion des demandes de fichiers 96
version 3.2, site web 19
Xerces 476
Apache Jakarta Project 152
bibliothèques de balises Taglibs et Struts 511
ECS, site web 153
moteur d'expressions régulières 525
WebMacro (clone Velocity) 429
Apache Software Foundation (ASF), paquetages javax.servlet et javax.servlet.http ix
Apache Struts (bibliothèque de balises) 511, 516
balise <iterate>, remplacement des boucles for 512
balise <property> 512
installation 512
API
côté serveur, dans J2EE 365
extension du serveur 4
Java API for XML Parsing (JAXP) 365
Java fondamentales, exploitation pour les servlets 10
JavaMail 523
JDOM, manipulation de XML avec 420
API Servlet 6
authentification, configuration 221
classes et méthodes, développement routinier de servlets 12
conteneurs de servlets supportant, liste 19
cookies, support pour 198
extensibilité de 12
fonctionnalités de suivi de session 203–218
SessionSnoop (exemple) 213
référence 563–591
servlets HTTP 16–18
sûreté du typage de 11
Version 2.1
mécanismes de partage d'informations 347
techniques d'abstraction des ressources 98
Version 2.2
changements par rapport à la version 2.0 x
classe ClassCastException 40
Version 2.3 547–561
attributs d'erreur, nouveaux 555–558
chargeurs de classes 554
clarification de la DTD 560
classes enveloppes pour les requêtes/réponses HTTP 550
codage de caractères, sélection 553
dépendances avec les JAR 554
événements du cycle de vie 551–553
filtres 548–551
version supportée par le serveur, obtention 80–82
VersionSnoop (exemple) 82
versions récentes ix
API spécifiques au serveur, utilisation de code C ou C++ 4

append() 544
applet chat, conception 328
applet Suivi d'actions (exemple) 287–293
  communication avec le serveur
    approche hybride 293
  données provenant d'un serveur 287
  sérialisation d'objets, utilisation 291
  serveur utilisant les rappels pour notifier
    une modification du cours 292
  valeurs des actions, obtention avec une
    invocation de méthode RMI 292
applets
  communication avec des serveurs 287–346
    applets sûres et non sûres 287
    clients se connectant à des serveurs 248
    combinaison HTTP, non-HTTP et
      RMI 293
    connexions HTTP et par socket 288–290
    CORBA-IIOP, utilisation 292
    JDBC, utilisation 291
    RMI, utilisation 291
    servlets et sérialisation d'objet 291
  communication avec des servlets 287–346
    applet chat, conception 328
    classe HttpMessage 303
    HTTP à base d'objets 304–307
    HTTP à base de texte 297–304
    HttpChatApplet (exemple) 334–337
    paramètres d'applet générées par une
      servlet 345
    RMI, utilisation 316–327
    RMIChatApplet (exemple) 338–343
    SocketChatApplet (exemple) 337
    sockets, non-HTTP 307–316
  exceptions, envoi à une servlet 307
  portabilité, servlets contre 10
  suivi de session avec 211
application de recherche de code régional
    (WML) 161–165
application de substitution, XMLC
  Hello.java code source autogénéré 472–476
Application, interface, implémentée par des
    applications Tea 413
application/octet-stream, type de contenu 104
application/x-www-form-urlencoded, type de
    contenu 111, 303
ApplicationRequest, interface 414
ApplicationResponse, interface 414
applications
  côté serveur, adéquation de Java 1
  JSP
    affichage des outils 506–511
      avec des bibliothèques de balises
        personnalisées 513–516
  Tea 410–416
    classe NameApp (exemple) 413
    classe NameContext (exemple) 414
    écriture 413–416
    fonctions de traitement du texte 410
    gestion des requêtes/réponses 412
    liste des outils disponibles 416–427
  WAP, exécution avec des simulateurs de
    périphériques 158
  WebMacro, affichage des outils 447–452
  XMLC
    affichage des outils 480–488
    substitution simple 470, 470–476
  (voir aussi applications web)
applications bancaires
  autorisation personnalisée à base de
    formulaire 233
  transactions 267
applications outils
  JSP
    fichier d'en-tête 507
    fichier de base de page 508
  XMLC
    classe manipulation 485–488
    fichier de contenu HTML 482–484
    fichier de modèle HTML 480–482
applications web 24–31
  classes, séparation des classes du serveur 554
  définition 24
  descripteurs de déploiement 26, 28–31
    ajout des correspondances de servlets 29
    référence de la DTD 621–649
    XML et DTD 27
    (voir aussi descripteurs de déploiement ;
      DTD)
  développement, avantages des servlets 10–
    12
  distribuables 363
  élément <web-app> 647
  entrées de l'environnement, contexte
    java:comp/env 367
  locales supportés, déterminaison 393
  pages d'erreur, configuration 141
  pages d'exception 147
  répertoire WEB-INF 25
  structure des fichiers 25
  suivi de session 203
  sûres, valeur de timeout pour 209
  synchronisation de l'état avec la base de
    données 552
  Tea 410–416
    vente sans le code source 409
  timeout de session par défaut,
    définition 207

architectures
distribution de servlets 364
JSP
de type Model 2 511
options, site web 490
servlet contrôleur Model 2 513–515
TeaServlet, WebMacro contre 430
archivage
fichiers application web (WAR) 25, 98
dépendances avec les fichiers JAR, expression dans 554
fichiers Enterprise Archive (.ear) 366
fichiers Java Archive (voir JAR, fichiers)
AreaCode, servlet (exemple) 161–164
arithmétique entière, outil du contexte WebMacro pour 437
ArrayIndexOutOfBoundsException, classe 111
ArrayList, classe 545
arrêt des threads en arrière-plan 52
arrière-plan, servlets (voir servlet d'arrière-plan)
arrondi des valeurs de date 55
ASP (Active Server Pages) 5
ATG Dynamo Application Server 8
attaque par saturation, empêcher 111
attributeAdded() 551
attributeRemoved() 551
attributeReplaced() 551
attributs
contexte de servlet, auditeurs du cycle de vie 551
définition dans des cookies 199
définition, obtention et suppression 348
expression d'attribut au moment de la requête 503
informations sur les erreurs 142
JSP, liste 495
<jsp:useBean>, action 500
noms, conventions 74, 121
requêtes
connexions HTTPS, API Servlet 2.3 558
obtention 121
requêtes d'inclusion 356
retrouver les éléments de chemin et les paramètres 359
requêtes de renvoi, utilisation dans 352
serveur 74
javax.servlet.context.tempdir 75
serveur et servlet, liste 74
XML, entourer des valeurs avec des guillemets 27
auditeurs
contexte de servlet, gestion des connexions à une base de données partagées 552
événements du cycle de vie d'une servlet 551
journalisation lors des liaisons/déliaisons à une session 216
liaison/déliaison des sessions 215
session 552
<auth-constraint>, élément 224, 627
<role-name>, caractère générique pour 560
authentification
avec fermeture de session
champs de formulaire cachés, utilisation 195
réécriture d'URL, utilisation 198
fabriques de ressources 370
HTTP 220–226
configuration 221
retrouver les informations sur 225
servlet Authentication (exemple) 226
servlet utilisant, ajout d'éléments au caddie 192
SSL 240
configuration 242
retrouver les informations sur 243–246
type, changements dans l'API Servlet 2.3 559
utilisation avisée 85
authentification à base de formulaire 227–230
pages d'ouverture de session et d'erreur, spécification 228
authentification basée sur les rôles 221
configuration 221
servlet Authentication (exemple) 226
authentification de base
contre basée sur les formulaires 230
envoi des informations avec HttpMessage 303
limitations 220
passer à l'authentification à base de formulaire 227
restriction de l'accès aux servlets dans web.xml 223
serveurs équipés avec SSL, utilisation 240
valeur BASIC, balise <auth-method> 224
authentification digest 221
valeur DIGEST, balise <auth-method> 224
authentification personnalisée 230–237
à base de formulaire 233–237
restriction des accès aux servlets 231–233
<auth-method>, élément 627
passer de BASIC à FORM 228
Authorization, en-tête 107, 231
autorisation
à base de formulaire 233
basée sur les rôles 221
contrôle avec les servlets 85
autorité de certification 239
autovalidation, mode 268

B

balisage
  contenu, basé sur les servlets 151–153
    outils de création 151–153
  langage de modèle de WebMacro 436
balises
  dans les fichiers web.xml, ordre des 29, 223
  HTML contre XML 27
balises d'ouverture/fermeture, contraintes dans XML 27
balises de fermeture, contraintes dans XML 27
balises vides, indication avec /> 502
barre de menu (page), modèle Tea 425
bas de page
  fichier footer.jsp 508
  modèle Tea pour 425
  modèle WebMacro pour 448
basculement de session 364
base de données des utilisateurs 222
base de données Java, Cloudscape 286
base de données, connexions (voir connexions à une base de données)
<BASE>, balise, HTML 529
Base64Decoder, classe, téléchargement 231
Base64Encoder, classe, téléchargement 303
bases de données ODBC
  accès avec le pont JDBC 252
  servlet OrderHandler (exemple) 268
bases de données relationnelles 249–251
  langages de programmation, procédures stockées 282
  pilotes JDBC 252
  SQL, mise à jour avec 251
    (voir aussi SQL)
  transactions 267–277
  (voir aussi connexions aux bases de données ; JDBC)
bcc() 523
BEA System WebLogic Application Server 8
beanName, attribut 501
beans entité, références à 368
beans session, références à 368
Beans, classe 501
#begin et #end, mots-clés, blocs de code pour les directives WebMacro 439
bibliothèque de balises personnalisées, JSP
  application outils utilisant 513–516
bibliothèques
  répertoire des classes WEB-INF/lib 25
bibliothèques de balises
  standard
    Apache Taglibs 511
    processus de demande de spécification Java 511
bibliothèques de balises personnalisées, JSP 511–516
  application outil utilisant 513–516
  installation et utilisation 512
bind(), étapes du processus 323
BLOB (binary large objects), support dans JDBC 2.0 286
blocage pour les nouveaux messages 333
blocs catch (construction try/catch/finally) 150
blocs de code
  démarcation pour les directives WebMacro 439
  Tea, chaînes entourées de guillemets doubles ou simples dans 405
  utilisation des objets partagés, synchronisation 266
blocs finally
  fermeture de la connexion applet/servlet par socket 316
  libération des ressources graphiques 169
  servlet libérant les ressources dans un 150
blocs try/catch
  exceptions SQL, interception 258
  JSP, difficultés avec 496
Body, classe 457
boucle
  gestion dans l'instruction foreach (Tea) 405
  sur les listes WebMacro 440
boucles for, remplacement par la balise <iterate> 512
broadcastMessage() 333, 337
  doPost(), enveloppe autour 333
BrowserHawk4J, produit (Cyscape) 183
BufferedImage, classe 168
BufferedReader, classe 110
BugIdCustomizer, classe (exemple) 462
buildDocument() 476
ByteArrayOutputStream, classe 144, 182
bytecode (Java), compilation des modèles Tea 408

C

CacheHttpServlet, classe 56–67, 182, 277
  code source 60–67
  méthodes, discussion 59
  modèles WebMacro, utilisation avec 452
  utilisation par Guestbook (exemple) 56–59
CacheHttpServletResponse, classe 59
CacheServletOutputStream, classe 59
CallableStatement, classe 283
canal sécurisé, spécification pour les cookies 200
caractère sur un octet 377
caractères ASCII 381
  dans de grandes chaînes de texte 284
  dans des lots de ressources 386

caractères de contrôle (SQL), gestion correcte 264
caractères ISO-8859-1 (Latin-1) 377, 659–664
caractères multioctets 377
caractères US-ASCII 381
  dans des lots de ressources 386
casse
  balises XML, sensibilité à 27
  mots-clés SQL 250
  noms de fichier des modèles Tea 401
Caucho Resin, serveur web 8
cause racine, objet Throwable 146
cc() et bcc() 523
certificats client
  authentification basée sur 242
  informations, retrouver avec les servlets 243
  spécification dans la balise <auth-method> 224
  X.509v3, servlet examinant 243–246
certificats numériques 220, 237–239
  clés publiques, stockage dans 238
  pour les clients 240
    authentification basée sur 242
CGI (Common Gateway Interface) 2
  cookies, envoi 200
  FastCGI contre 3
  indépendance de la plate-forme 4
  information de chemin supplémentaire, besoin 93
  langage Perl, utilisation de 2
  outil du contexte de WebMacro 437
  PerlEx, amélioration des performances 3
  schéma de requête 106
  scripts sur le serveur, applets effectuant des connexions HTTP à 288
  serveurs, manque d'interaction avec 70
  servlet retrouvant les informations d'environnement contre 69
  types de données, manque de sûreté dans 11
chaînage
  filtres 549
  méthodes 457
  servlets 13
chaînes
  concaténation dans Tea 405
  guillemets, utilisation avec caractères de contrôle SQL et bases de données 264
chaînes d'interrogation 15
  applets utilisant pour les requêtes 289
  codée comme URL, création à partir de la liste de propriétés 302
  formulaire POST avec une URL d'action contenant 89
  requêtes, retrouver 88
  servlet Hello, utilisation dans 21
chaînes de caractères
  nulles, comparaison avec une autre chaîne 404
  paramètres d'initialisation sous forme de 44
  paramètres d'un objet modèle (WebMacro), obtention sous forme de 441
  paramètres de requête, renvoi comme 519
  sortie, modèles WebMacro 432
  texte converti en octets, construction à partir de 398
  traces de la pile, obtention sous forme de 144
  variables WebMacro 439
champs de formulaire cachés
  charset 394–398
  suivi de session avec 193–195
CharacterFilter, classe 459
charge, distribution 361–365
  architectures standard, liste 364
  servlets d'entreprise
    règles 362
chargement avec la directive taglib 516
chargement d'une image en mémoire 174
chargement de classes
  à partir des répertoires WEB-INF/classes et WEB-INF/lib 25
  application web, séparation du serveur 554
  applications Tea 416
  dans le chemin de classes du serveur 40
  partage d'objets entre contextes, problèmes 350
  pilote JDBC 253
  rechargement de servlets 39
  rechargement, problèmes 544
  stub et squelette, RMI 318
chargement des servlets au démarrage du serveur 52
CharsetAction, servlet (exemple), conversion de charset natif en Unicode 396–398
charsets 665–666
  API Servlet 2.3, support additionnel 553
  classe LocaletoCharsetMap 388
  classe ParameterParser, spécification pour les requêtes 518
  définition automatique pour la réponse de la servlet 393
  formulaires HTML avec du contenu localisé, suivi 394–398
  langues non occidentales 377
  langues occidentales 377
  navigateurs, utilisation de 377
  par défaut pour les réponses 125
  préférences, indication 385
  servlets, utilisation de 378–381
CharToByte, classes de conversion 379

Chat Server/Servlet, service complet (exemple) 329–334
ChatClient, interface, implémentée par RMIChatApplet 338–343
ChatDispatch, servlet (exemple) 343–346
ChatServer, interface (exemple) 329
ChatServlet, classe (exemple) 329
checkError() 150
chemin racine du document 94
chemins absolus
  TemplatePath, inclusion d'un fichier cible #parse 440
  URL, distribution de requêtes 351
chemins de classes (voir information de chemin)
chemins relatifs
  à la racine du contexte courant 351
  distribution de requêtes 351
  TemplatePath, inclusion d'un fichier cible #parse 440
chemins virtuels 92
  retour du chemin réel 94
chiffrement fort, politique d'export du gouvernement U.S. 83
chinois, langue, charset pour 377
-class, option, XMLC 475
ClassCastException, classe 40, 325, 350, 501, 504
classe, variables de 38
classes
  API Servlet 6
    référence 563–591
  application, Tea 413
    état, invisibes aux modèles 423
  bibliothèque de balises personnalisées, installation 512
  chargement
    classes d'application, Tea 416
    classes dans le chemin de classes du serveur 40
    partage d'objets entre contextes 350
    pilote JDBC 253
    rechargement de servlets 39
    stub et squelette RMI 318
  com.oreilly.servlet xiii
    CacheHttpServlet 277
    DaemonHttpServlet 311–314
    LocaleNegotiator 388–393
    ServletUtils 97, 99, 144
    VersionDetector 80–82
  courrier, envoi d'e-mail avec 523
  HTML 456
  Java
    compilation d'une page HTML en 470
    introspection 500
  JavaBeans, spécification du nom de 501
  manipulation, création d'une instance de l'arbre 469
  rechargement, problèmes 544
  séparation de l'application web et du serveur 554
  WEB-INF/classes et WEB-INF/lib, répertoires 25
classes de manipulation, XMLC 476–480
  maquette de liste de langues 478–480
  page Hello 476
  ToolViewServlet (exemple) 485–488
classes stub et squelette, RMI 292
  chargement 318
  compilateur rmic, génération avec 318
  DaytimeServlet (exemple) 318
  génération pour RMIChatApplet 343
classes utilitaires globales, installation dans le chemin de classes de la servlet 40
classes utilitaires, paquetage com.oreilly.servlet xiii
ClassNotFoundException, classe 258
clé de licence, génération pour les clients 90–92
clé publique, cryptographie 237–239
Clear Ink, bibliothèque Village à sources ouvertes 286
clearParameters() 264, 266
clés, cryptographie
  clé publique/privée 237
    pour l'algorithme RSA 238
  symétrique, utilisation dans SSL 239
  test avec une taille de 128 bits 558
  vérification, autorités de certification 239
clients
  authentification, SSL 240
  de discussion
    connexions par socket non-HTTP 333
    HTTP 328, 333
    notification de nouveaux messages 333
    RMI 329
  déconnexion des servlets 149
  informés des erreurs 140
  Java, restriction d'utilisation de RMI 292
  obtention des informations sur 82–121
    accueil personnalisé, création 86
    attributs supplémentaires 121
    clé de licence, génération 90–92
    déterminer ce qui a été demandé 104
    en-têtes de requête 106–110
    flots d'entrée (requête) 110–121
    information de chmein 92–95
    lecture à partir d'une ressource abstraite 98–100
    machine du client 82
    requêtes 87–90
    restriction de l'accès 83
    servir des fichiers 96–98
    servir des ressources 100–104
    types MIME 95

utilisateurs 84–85
ouverture de session 192
RMI, servlets agissant comme 534
RMI, socket et HTTP, support du serveur d'applications 293
s'identifiant auprès des serveurs 191
utilisation des formulaires HTML ou des applets, connexion à des serveurs 248
(voir aussi sérialisation d'objet ; serveurs ; servlets)
CLOB (character large objects), support dans JDBC 2.0 286
close() 125
Cloudscape, base de données 286
clustering 364
serveurs, niveau de support du 365
Cocoon (création de contenu basé sur XML) 153
codage
Base64, nom d'utilisateur et mot de passe, décodeur pour 231
contenu compressé 183
images graphiques 182
codage à longueur variable, jeu de caractères UTF-8 381
codage d'URL, formulaire HTML avec du contenu localisé 394
codage de caractères
API Servlet 2.3, support additionnel 553
formulaires HTML avec du contenu localisé 394
inconnu ou non supporté 110, 125, 379
lecture et écriture, options 380
(voir aussi charsets)
code
blocs synchronisés 36
des exemples, téléchargement xiii, 13
modèles Tea 401
code de tiers, classe Java et composants JavaBeans 11
code natif, exécution à partir de Java
programme finger 532
code source
autogénéré, classe Hello 472–475
classe CacheHttpServlet 60–67
classe ParameterNotFoundException 522
classe ParameterParser 519–522
exemples du livre, téléchargement xiii, 13
servlet, page JSP 492
code, paramètre 161
codes d'état 651–657
code Not Found 124
définition 123
ErrorDisplay (exemple) 148
errorTaker.jsp (exemple), utilisant une scriptlet 497
gestion des erreurs avec 143
Not Modified (304) 55
rapport des erreurs avec 140
réponse HTTP 14, 129–132
définition 131
ViewFile, amélioration avec 132
SC_NO_CONTENT 333
SC_SERVICE_UNAVAILAVLE 147
SC_UNAUTHORIZED 231
servlet utilisant dans l'authentification de base 230
servlets de renvoi, définition 353
servlets incluses et 359
codes de pays 375
codeurs, formats d'image 168
collaboration de servlets 347–359
par manipulation directe 347
partage d'informations 347–351
interface ServletContext 348–351
requêtes, partage du contrôle 351–359
collections
fichier JAR pour 433
itération sur 405
balise <iterate> 512
colonnes (ensemble de résultats), retrouver les valeurs des 257
combinaison d'images (pour créer un compteur graphique) 175–178
commentaires
cookies 200
JSP 494
contenant un balisage HTML 516
Tea et Java 406
WebMacro, syntaxe 433
commit() 268
Common Gateway Interface (voir CGI)
communication applet/serveur HTTP à base d'objet 291
communication applet/serveur HTTP à base de texte 297–304
communications client/serveur, adéquation des servlets 11
communications sûres
à partir d'une applets 303
connexions HTTP applet/serveur 289
compilateurs
javac 20
rmic 318
xmlc 470
compilation
fichiers modèle Tea 408
servlets 20
Component, classe 168, 172
compress (ou x-compress), schéma de codage 183

compressé, contenu 183–186
en-tête Content-Encoding, indication dans 185
envoi, ViewResourceCompress (exemple) 184–186
compte à rebours, utilisation du push du serveur pour l'affichage 187–189
compteurs
généraux (exemple) 38–39
graphiques, combinaison d'images pour créer des 175–178
InitCounter (exemple), lecture des paramètres d'initialisation 42
InitDestroyCounter (exemple) 44–47
SimpleCounter (exemple) 35
suivi de session, utilisation pour le comptage des hits 204
synchronisés simples 35
com.sun.*, noms d'attributs 121
com.sun.*, paquetages 74
com.sun.image.codec.jpeg, paquetage 169
concaténation de chaînes
ajout par, performances des servlets et 544
langage Tea 405
conf/tomcat-users.xml, fichier 222
confidentialité
cryptographie, garantir avec 238
garantir pendant le transit 242
Confidentializer, servlet (exemple) 172–175
configuration
authentification à base de formulaire 228
authentification HTTP 221
basée sur des rôles 221
restriction des accès aux servlets 222–225
authentification SSL 242
composants HTML 457
dans les descripteurs de déploiement 26
motifs d'URL, correspondances de servlet 29
(voir aussi descripteurs de déploiement)
fichier racine_serveur/conf/server.xml, ajout de nouvelles correspondances de contexte 26
filtres 549–551
pages d'erreur du serveur 141–143
pages d'exception 147
sécurité SSL 241
TeaServlet 407–408
timeouts de session individuels 207
WebMacro, dans les fichiers de propriétés standard 433–435
Connection, interface 253, 256
ConnectionHolder, classe (exemple) 274
ConnectionPerClient, servlet (exemple) 274–277
ConnectionPool, classe 271–273
connexions à une base de données 247–286
API Village et Town 286
bases de données relationnelles 249–251
classe DaytimeConnection (exemple) 310
classe ResultSetBuilder, API JDOM 420
ensembles de résultats, requêtes SQL 259–261
établissement avec les paramètres d'initialisation d'une servlet 71
exceptions SQL, gestion 258
gestion avec SingleThreadModel 48–50
JDBC 251–264, 282–285
basées sur des fichiers de propriétés 254
champs null, gestion 261
chargement du pilote et création de la connexion initiale 253
communications applet/serveur 291
grands types de données, gestion 284–285
mise à jour de la base de données 262
obtention à partir des servlets 255
procédures stockées 282
Version 2.0, fonctionnalités supplémentaires 285
partagées, auditeur de contexte gérant 552
réserve 270–274
servlet OrderHandlerPool (exemple) 273
réserve externe, gestion avec 50
réutilisation 265
servlet Livre d'or (exemple) 277–282
timeouts de session et 209
transactions 267–277
JDBC, utilisation avec 268
liaison à 269
suivi de session, gestion avec 274–277
connexions keep-alive (voir connexions persistantes)
connexions par socket
classes URL et URLConnection, utilisation pour 302
client/serveur
pull du client et 186
push du serveur et 186
clients de discussion utilisant 337
pour poster des messages 328
clients de discussion, communiquant avec 328
communication applet/serveur
à des serveurs non-HTTP 289
avantages et inconvénients 290
non-HTTP, communication applet/servlet 307–316, 326
clients de discussion, notification de nouveaux messages 333

combinaison avec HTTP et RMI 293
DaemonHttpServlet (exemple) 311–314
DaytimeApplet (exemple) 314–316
DaytimeServlet 308–311
persistantes 126
protocole Secure Sockets Layer (voir SSL)
serveur, établissement/libération avec les threads démon 313
servlets, établissement avec un serveur de courrier 523
connexions persistantes 126
servlet utilisant la mise en tampon pour supporter 127
constantes String statiques finales (types d'authentification, API Servlet 2.3) 559
conteneur de servlets additionnels
liste 9
conteneurs
Enterprise JavaBeans (EJB) 362
servlet
fonctionnalité de clustering 364
séparation de l'application web des classes du serveur 554
(voir aussi serveurs)
conteneurs de servlet embarquables 9
conteneurs de servlets
choix 7, 10
contraintes du cycle de vie 33
existants, liste avec le niveau de support de l'API 19
getHeaderNames(), n'implémentant pas 109
plug-ins, débogage 538
conteneurs de servlets additionnels 7, 9
conteneurs de servlets autonomes 7
conteneurs de servlets embarquables 7
Content-Encoding, en-tête 183, 185
Content-Language, en-tête 373, 387, 393
Content-Length, en-tête 126
Content-Type, en-tête 24, 125, 126, 378, 387, 393
contenu
création basée sur les servlets 151–153
outils pour 151–153
défini 151
fonctions de traitement, Tea 412
modèle (architecture MVC) 151
WAP
dynamique 160–165
servir 159–160
contenu dynamique
CGI, utilisation 2
génération avec JavaScript côté serveur (SSJS) 5
graphique, génération 169–171
inclusion directe, JSP 492
internationalisation 371–398
langues multiples sur une page 381–384
négociation de la langue 384–393
préférences de charset 385
obtention de la date de dernière modification 55
WAP 160–165
contenu multimédia
envoi 155–189
compressé 183–186
images 165–183
push du serveur, utilisation 186–189
images
composition 172–178
effets spéciaux 178–183
génération 166–172
contenu statique, génération par la servlet d'arrière-plan pour les JSP 492
contextDestroyed() 551
contexte
chemin, obtention 95
courant, examen de la page d'accueil 100
initialisation et destruction, auditeurs 551
non-Web, retrouver 431
objets passés à un modèle WebMacro 431
par défaut, obtention 100
problèmes ClassCastException et 40
ressources d'un autre, obtention 100
(voir aussi ServletContext, interface)
contexte, paramètres d'initialisation 78–80
connexions à une base de données, définition 256
informations de connexion à une base de données 255
recherche du registre RMI avec 79
contextInitialized() 551
<context-param>, élément 78, 627
ContextProperties, classe 255
servlet Livre d'or, utilisation 277
spécification des informations de configuration de la base de données 281
contrôle, partage entre servlets 351–359
contrôleur (architecture MVC) 151
conversion de chemins
API Servlet 2.3, changements 560
non passés comme information supplémentaire 94
passés comme information supplémentaire vers le système de fichiers 93
Cookie, classe 198, 594–597
cookies
détection de l'identifiant de session 213

envoi avec HttpMessage 303
envoyés dans une requête, affichage avec des balises personnalisées 512
identifiant de session, stockage dans 211
outil WebMacro pour 437
persistants, suivi de session avec 198–203
caddie virtuel (exemple) 201–202
obtention 199
remplacement par la réécriture d'URL 212–213
spécification de l'âge maximum 200
CORBA (Common Object Request Broker Architecture)
dans J2EE 365
Inprise Application Server 4 8
objets distribués dans des langages différents, communication entre 292
coréen, langue, charset pour 377
corps de la réponse 124
effacement 560
correspondance de motifs
domaines, pour les cookies 199
élément <url-pattern> 647
instances de servlet et 38
servlets, correspondance avec des URL 28–31
URL, spécification des accès restreints 224
(voir aussi expressions régulières ; motifs d'URL)
correspondance octets à caractères, langues occidentales 377
côté serveur, API, J2EE 365
côté serveur, applications, langage Java 1
couche application, WAP 157
couche de transport
protocoles, réseaux sans fil 155
RMI, fonctionnement au travers des firewalls 292
couleur, image, conversion en niveaux de gris 179–181
count, paramètre, obtention de la valeur comme un int 517
createContext() 413, 414, 420
createDocument() 476
createImage() 168, 172
createStatement() 256
createTempFile() 76
createTextNode() 480
création de sessions 552
cryptographie 237–239
algorithme de chiffrement MD5 221
certificates numériques, utilisation 220
clés publiques/privées 237
clés symétriques, algorithme DES 238
RSA, algorithme de chiffrement 238
SSL, utilisation 239
cryptographie à clé privée 237
cryptographie à clé publique 237–239
RSA, algorithme de chiffrement 238
SSL, utilisation 239
cryptographie à clés publique/privée asymétriques 238
currentTimeMillis() 498
CustomBindingListener, classe 217
cycle de requête/réponse (serveur générique), classes pour 17
cycle de vie
applications Tea 413
connexions à une base de données, gestion 274–277
servlets 33–67
chargement au démarrage 52
compteur lisant des paramètres d'initialisation (exemple) 42
compteur synchronisé simple 35
événements ajoutés dans l'API 2.3 551–553
HolisticCounter (exemple) 38–39
init() et destroy() 40–47
mise en cache côté client 54–56
mise en cache côté serveur 56–67
modèle mono-thread 48–50
rechargement 39
SimpleCounter (exemple) 35
traitement en arrière-plan 50–52
une seule JVM 34
sessions 206
méthodes pour la gestion 209
cycle requête/réponse, HTTP
applications Tea 414
classes enveloppes, API Servlet 2.3 550
communication applet/programme CGI, problèmes avec 289
cookies, travail avec 199
filtres 548–551
objets de requête/réponse, retrouver à partir de WebContext 432
outils du contexte de WebMacro pour 437
requête/réponse WAP, conversion en 157, 164
RMI, remplacement avec 291
simulation d'une communication bidirectionnelle avec 328
Tea, fonctions de gestion 412
cyrillique (russe), alphabet, charset pour 377

D

DaemonHttpServlet, classe (exemple) 308, 311–314, 326
Data Encryption Standard (voir DES)
DataInputStream, classe 338

date
  dernière connexion d'un client, conserver 86
  fichiers, dernière modification 182
  serveur Daytime 293–327
  timeouts de session 207
date de dernière modification (voir modification/accès, date)
dates/heures
  affichage, personnalisation avec ECS 461
  localisation des formats 375
    espagnol 375
    français 412
    japonais 379
  modèle WebMacro, sortie avec 433
Daytime, serveur (exemple) 293–327
  interface DaytimeServer 316
DaytimeApplet (exemple) 294–327
  code 294–296
  fichier HTML référençant 296
  interface utilisateur 297
  obtention de l'heure avec HTTP
    basée sur du texte 298–304
    basée sur les objets 304–307
    classe HttpMessage 299–304
  obtention de l'heure avec RMI 324–327
  obtention de l'heure avec une connexion par socket 314–316
  retrouver une référence à un serveur distant 325
DaytimeClientServlet (exemple) 535
DaytimeConnection, classe (exemple) 310
DaytimeServlet (exemple)
  communication par socket non-HTTP 308–311
  interface DaytimeServer, implémentation 316
  support des accès HTTP de base 297
  support des accès RMI 317–324
    classe mère RemoteHttpServlet 320–324
    compilation avec rmic 318
  utilisation de HTTP pour servir un objet 304
DBGifReader, classe (exemple) 284
DBPhoneLookup, servlet (exemple) 257–258
DBPhoneLookupReuse, classe (exemple) 265
débogage
  log(), utilisation 143
  messages d'erreur dans les JSP 499
débogage de servlets 536–544
  ajustements des performances 544–546
  débogueur standard, utilisation 537
  examen de la requête du client 538–540
  journaux, contrôle 536
  navigateurs, utilisation dans 544
  outil tiers, utilisation 543
  requête de client personnalisée, création 540–543
débogeurs d'IDE, problèmes avec 538
déchargement de fichiers, à l'aide des flots d'entrée 111–121
  listes des fichiers déchargés 114
déclarations JSP 494
  <%! %>, délimiteurs de balisage 494
DeColorize, servlet (exemple) 178–180
  mise en cache de l'image convertie 182
défaire des transactions 267
DefaultHTMLDomFactory, classe 476
DefaultServlet, classe (Tomcat) 132
  écriture de replacement 96
deflate, schéma de codage 183
DELETE, instruction SQL 251, 262
DELETE, méthode HTTP 16, 17
-delete-class, option XMLC 478
démarrage des serveurs 20
DES (Data Encryption Standard) 238
  chiffrement sur 56 bits 242
  clés sur 128 bits, utilisation 238
  SSL, utilisation dans 240
descripteurs de déploiement 26
  ajout des correspondances de servlets 29
  alias de rôle dans 225
  auditeurs, déclaration dans 552
  authentification
    basée sur les rôles 221
    certificats client, basée sur 242
    de base, passage à celle basée sur un formulaire 227
  balise <distributable/> dans 363
  balises, ordre imposé dans 223
  contenu de type WAP, ajout à 159
  correspondances d'URL, ordre de recherche 30
  correspondances extension de fichier à type MIME, configuration 95
  DTD pour l'API Servlet 2.2 621–649
  DTD pour l'API Servlet 2.3 548, 560
  élément <filter> 550
  entrée <taglib>, ajout pour les bibliothèques de balises personnalisées 512
  entrées de l'environnement 366–368
  exemple 28–31
  <load-on-startup>, ajout à l'entrée <servlet> 52
  pages d'erreur, configuration dans 141
  pages d'exception, spécification 147
  paramètres d'initialisation d'une servlet, définition dans 41

paramètres d'initialisation du contexte 78
définition pour les connexions à une base de données 256
références de composant EJB 369
requêtes WebMacro, enregistrement de servlet pour 445
ressources nommées, distribution vers 352
sécurité SSL, spécification dans 241
TeaServlet, enregistrement 402
timeout de session, définition de la valeur par défaut 207
XML et DTD 27
<description>, élément 367, 628
dessin au-dessus d'une image existante (exemple Confidentializer) 172–175
destroy() 40–47
classe GenericServlet 44
gestion des exceptions 146
implémentée dans une servlet distante, appel de super.destroy() 317
interface Application 413
libération des ressources et sauvegarde des informations dans le cache 42
RemoteDaemonHttpServlet (exemple) 326
saveState(), appel à partir de 47
serveur appelant une servlet 145
servlets d'arrière-plan, JSP 495
servlets, écoute des applets 308
thread d'arrière-plan, arrêt 52
destruction
contextes de servlet et sessions 551
sessions 552
threads 313
développement d'applications d'entreprise, J2EE 365
digest, authentification 221
directives 433, 494
JSP 495–500
include 504
pages d'erreur, utilisation 497–499
syntaxe 495
taglib 513, 516
WebMacro 439–441
disponibles aux modèles, affichage 442
#else 439
#foreach 440
#if 439
#include 440, 441
#parse 440
#set 433, 439
#use 441
display() 460
<display-name>, élément 628
distribuées, transactions, JDBC 2.0 286
<distributable>, élément 363, 628
Distributed Application Tester (DAT), outil de débogage 543
distribution de la charge 361–365
architectures standard, liste 364
servlets d'entreprise
dans des environnements J2EE 370
règles d'écriture 362
distribution des requêtes 351–359
à des ressources spécifiées par un nom 352
inclusions 355–359
renvois 352–355
redirections contre 355
<DIV>, élément, fichier de maquettage HTML 482
Document Type Definitions (voir DTD)
documents bien formés 27
doDelete() 17
doFilter() 549
doGet() 17, 19
appel de saveState() 47
cache côté client et 54
choix de l'implémenter 23
classe ChatServlet (exemple) 333
classe PrimeSearcher (exemple) 52
distribution des requêtes POST à 23
en-tête Last-Modified, définition dans 55
exceptions WebMacro, interception et traitement 432
gestion des exceptions 146
mot-clé synchronized, ajout 37
requêtes HEAD dans 23
réutilisation des objets de base de données 265
servlet Guestbook (exemple) 281
servlet pilotant les modèles WebMacro 431
ToolViewServlet (exemple) 487
DOM (Document Object Model)
analyseurs XML supportant différentes versions 554
HTML, conversion en 470
interface HTMLDocument 475
nœuds 487
problèmes avec 480
représentation arborescente du XML 469
DOMFormatter, classe 477
données binaires
écriture sur un flot de sortie 125
gestion par la base de données 284–285
lecture à partir d'un flot d'entrée 110
structure requête/réponse, WAP 155
données caractères
lecture à partir d'un flot d'entrée 110
réponses, écriture 125
doOptions() 17

doPost() 17
choix de l'implémenter 23
classe ChatServlet (exemple) 333
gestion des exceptions 146
logique transctionnelle d'une commande 269
réutilisation des objets de base de données 265
servlet Guestbook (exemple) 281
synchronisation pour les transactions 270
doPut() 17
doTrace() 17
Driver, interface 252
DriverManager, classe 253
DTD (Document Type Definition) 26
descripteur de déploiement de l'application web
API Servlet 2.2 621–649
API Servlet 2.3 548, 560
documents XML et 27
fichier web.xml, ordre des balises dans 29
(voir aussi descripteurs de déploiement)
durée d'indisponibilité d'une servlet 147
durée des requêtes, filtre de journalisation 550
Dynamo Application Server 8
dysfonctionnement (serveur), échec des sessions et 364

E

.ear (Enterprise Archive), fichiers 366
échappement de caractères
automatique, élément XMLC 488
(voir aussi Unicode, échappement de caractères)
échappement des valeurs entre guillemets, PreparedStatement 264
éditeurs, XML 28
effets spéciaux, sur les images 178–183
mise en cache des images converties 181–183
EJB (Enterprise JavaBeans)
composants, référence via des entrées de l'environnement 368
J2EE 1.3, identité de sécurité de l'appelant 548
support de J2EE 11
<ejb-link>, élément 629
<ejb-ref>, élément 368, 629
<ejb-ref-name>, élément 630
<ejb-ref-type>, élément 630
Element Construction Set (ECS) 153, 455–468
composants de page comme des objets 455–457
ensembles de résultats, affichage 457–468
classe BugIdCustomizer (exemple) 462
classe DateCustomizer (exemple) 461
classe NumberCustomizer (exemple) 461
classe ResultSetTable (exemple) 463–464
classe TableCustomer (exemple) 459
frontal HTML, ResultSetServlet 467
ResultSetServlet (exemple) 464–466
SimpleResultSetTable (exemple) 457–459
éléments dans les fichiers web.xml, ordre 29
éléments de catalogue, inclusion de l'affichage dans la réponse courante 356–359
éléments de maquette, suppression pendant la compilation XMLC 478
elements() 448
#else, directive WebMacro 439, 443
e-mail, classes 523
embarquement de JavaBeans dans des pages JSP 500–502
encode() 169
encodeRedirectURL() 212
encodeURL() 212
#end, mot-clé, blocs de code WebMacro 439
endResponse() 188
Enhydra Application Server 8, 469
enregistrement
servlet MacroView, gestion des requêtes *.wm 445
servlets 28
TeaServlet, fichier web.xml 402
utilisateurs, suivi de session et 193
(voir aussi RMI, registre)
enregistrement des problèmes de réponse des servlets 140
ensembles de résultats 259–261
affichage avec ECS 457–468
classe BugIdCustomizer (exemple) 462
classe DateCustomizer (exemple) 461
classe NullCustomizer (exemple) 460
classe NumberCustomizer 461
classe ResultSetTable (exemple) 463–464
classe TableCustomizer (exemple) 459
ResultSetServlet (exemple) 464–466
ResultSetServlet, frontal HTML 467
SimpleResultSetTable (exemple) 457–459
défilants et modifiables, JDBC 2.0 286
génération de tables HTML à partir de, utilisation des métadonnées 259
génération de tables HTML avec ResultSetMetaData 262
méthodes pour retrouver les données 260
Enterprise Archive (.ear), fichiers 366

Enterprise JavaBeans (EJB) 362
utilisation comme tiers intermédiaire se connectant à une base de données 249
en-têtes
Accept-Charset 385
Accept-Encoding 183, 185
Accept-Language 385
accès aux valeurs 108
authentification, servlet utilisant dans 230
Authorization 107, 231
Content-Encoding 183, 185
Content-Language 373, 387, 393
Content-Length 126
Content-Type 24, 125, 126, 378, 387, 393
cookies, envoi avec 198
définition avec un objet HTTPServletResponse 19
e-mail, configuration 523
envoi avec HttpMessage 303
fichier header.jsp 507
HTTP 14
réponses 124
requêtes 106–110
servlets incluses et 359
If-Modified-Since 54
Last-Modified 54, 55
Last-Modified et If-Last-Modified 107
modèle Tea pour 425
modèle WebMacro pour 448
Retry-After 147
servlet HelloBabel (exemple) 387
servlets de renvoi, définition 353
User-Agent 183
Vary 185
WWW-Authenticate 107, 231
entiers
arithmétique, outil du contexte WebMacro 437
paramètres d'un objet modèle (WebMacro), obtention sous forme de 441
variables WebMacro 439
entités caractères 372
référence 659–664
entités caractères nommées 372, 659–664
entités caractères numériques 372, 659–664
exemple HelloSpain, utilisation dans 373
entrée, mise en tampon pour améliorer les performances des servlets 545
entrées de l'environnement 366–368
contexte java:comp/env, placement dans 367
élément <env-entry> 366, 630
élément <env-entry-name> 631
élément <env-entry-type> 367, 631
élément <env-entry-value> 367, 632
références à des composants EJB 368
références à des fabriques de ressources externes 369
Entrust Technologies (autorité de certification) 239
environnement auteur graphique (Kettle), Tea 428
environnement de publication web 153
environnements de développement d'applications web
ECS (Element Construction Set) 455–468
JSP (JavaServer Pages) 489–516
Tea 399–428
WebMacro 429–453
XMLC 469–488
envoi d'URL 15
equals() 439
équivalence, vérification 403
erreur, pages
configuration 141–143
erreurs
compilation, JSP 494
compilation, TeaServlet 408
gestion 139–151
exécution, Tea 410
journalisation 143
modèles WebMacro 446
page d'erreur affichée par une servlet 142
rapports aux clients 144
rapports avec les codes d'état 140
servlets envoyant une sortie à des clients inexistants 149
images, renvoi 175
messages de débogage dans les JSP 499
modèles Tea, affichage dans l'outil de création Kettle 428
pages d'erreur, serveurs 132
paramètre count, conversion en entier 517
erreurs d'exécution, gestion dans Tea 410
<error-code>, élément 632
ErrorDisplay, servlet (exemple) 148
codes d'état d'erreur et d'exceptions, affichage 148
trace de la pile (exceptions), non-disponibilité 149
errorMaker.jsp (exemple) 497
<error-page>, élément 632
errorTaker.jsp (exemple) 497
espaces
fichier web.xml, changements dans l'API Servlet 2.3 560
JSP, conservation obligatoire 498

espagnol (langue)
date et heure, localisation 375
HelloSpain (exemple) 372
date localisée, affichage 375
échappements Unicode, utilisation 374
entités caractères numériques, utilisation 373
establishConnection() (méthode personnalisée) 71
estampilles (voir modification/accès, date)
état
classes d'application (Tea) 423
environnements distribués, détention dans une ressource externe 362
HTTP, problèmes avec 191
servlets, sauvegarde entre les chargements 44–47
(voir aussi persistance)
événements, liaison de session 215–217
Exception, classe 146
exception, gestion
administration de Tea 410
exceptions
applet envoyant à une servlet 307
paramètres de requête, obtention et conversion 518
rapport, traces de pile et 144
servlet OrderHandler (exemple) 269
SQL, gestion 258
ToolServlet (exemple), gestion des 452
trace de pile et URI de la ressource problématique 556
WebMacro, gestion des 432
exceptions distantes, déclaration pour les méthodes RMI 317
<exception-type>, élément 633
exec()
ajout d'un paramètre à 532
classe Runtime 530
servlet Finger (exemple) 532
execute() 262
problèmes de sûreté vis-à-vis des threads 266
executeQuery() 256
executeUpdate() 262, 264
exécution de programmes externes à partir des servlets 529–533
programme finger 530–533
exemples
applet Suivi d'actions 287–293
application de recherche de code régional (WML) 161–165
application outil, WebMacro 447–452
BugIdCustomizer 462
Chat Server/Servlet, service complet 329–334
classe Buffering 129
classe CacheHttpServlet 60–67
classe Hello, code source autogénéré 472–475
classe HelloBean 504
classe HtmlSQLResult 262, 265
classe LocaleNegotiator 388–393
classe LogFilter 550
classe MultipartRequest 112–121
code 116–121
classe MultipartResponse 187–189
classe NameApp 413
classe NameContext 414
classe ResultSetTable 463–464
classe SnoopManipulation 478
classe SocketWatch 538–540
classe Tool 421–423
classe ToolApp 416–420
classe ToolBean 510
classe ToolContext 420
communication DaytimeApplet/DaytimeServlet 293–327
ConnectionHolder 274
DaemonHttpServlet 311–314
DaytimeServlet, communication par socket non-HTTP 308–311
DBPhoneLookup 257–258
drinks.xml (application WML) 157–158
fichier de propriétés de WebMacro 434–435
HeaderSnoop 109
HelloBabel 386–388
HelloJapan 378
HelloJapanReader 380
HelloRosetta 382–384
HelloSpain 372
date localisée 375
échappements Unicode, utilisation 374
entités caractères numériques 373
HelloWorldGraphics 167
HolisticCounter 38
HtmlResultSet 259
HttpChatApplet 334–337
HttpClient 541–543
InitDestroyCounter 44–47
KeyedServerLock 76–78
déverouillage 90–92
LoginHandler 236
ManualInvalidate 210
MathTool (WebMacro) 437
MyConnectionManager 553
NileBooks 356–358
NullCustomizer 460
OrderHandlerPool 273–274
page errorMaker.jsp 497
page errorTaker.jsp 497

page toolview-tag.jsp 515
pages d'erreur, configuration 147
PersonalizedWelcome 86
PrimeSearcher 50–52
ResultSetServlet 464–466
frontal HTML pour 467
RMIChatApplet 338–343
ServerSnoop 74
servlet CharsetAction 396–398
servlet ChatDispatch 343–346
servlet Confidentializer 172–175
servlet ConnectionPerClient 274–277
servlet DBGifReader 284
servlet DeColorize 178–180
servlet ExportRestriction 83
servlet FileLocation 94
servlet Finger 530–533
servlet HTTP Bonjour tout le monde 18–20
servlet InitCounter 42–44
servlet KeepAlive 127
servlet Livre d'or 277–282
servlet MacroView (WebMacro) 444–446
servlet OrderHandler 268
servlet ProtectedResource 235
SessionSnoop 213
SessionTimer 207
SessionTracker 204
ShoppingCartViewerCookie 201–202
ShoppingCartViewerHidden 193–195
ShoppingCartViewerRewrite 196–198
ShoppingCartViewerSession 217
SimpleChart 169–171
SimpleCounter 35
lecture des paramètres d'initialisation 42
snoop.wm (modèle WebMacro) 442
SpecialSetter 349
ToolServlet (WebMacro) 450, 450–452
toolview1 (modèle Tea) 423
toolview2 (modèle Tea) 423
ToolViewServlet 485–488
UploadTest 114
VersionSnoop 82
ViewFile
codes d'état, utilisation 132
journalisation des erreurs 143
trace de la pile d'une IOException 144
ViewResource 102, 165
ViewResourceCompress 184–186
WMServletHello 432–433
exit() 24
appel accidentel 140
expiration
cookies 200, 206
sessions 206
exportation des objets distants 322
exportation du chiffrement fort, politique du gouvernement U.S. 83
expression de passe, pour les clés privées 238
expressions JSP
affichage de valeurs avec (toolview.jsp) 510
attributs au moment de la requête 503
déclarations et 494
points-virgules, inutilisés dans 499
expressions régulières 525–529
balise <A HREF>, extraction d'une page 529
balise <BASE>, recherche 529
extensibilité et souplesses des servlets 12
Extensible Markup Language (voir XML)
<extension>, élément 633
extensions du serveur 4
écrites en C++, problèmes de gestion des exceptions 12
servlets en tant que 5

F

fabriques de ressources définies par l'utilisateur 370
fabriques de ressources externes, références à 369
FastCGI 3
FastWriter, classe 431
fenêtres, affichage 168
fermetures de session
authentification avec
champs de formulaire cachés, utilisation 195
réécriture d'URL, utilisation 198
authentification de base et à base de formulaire, manque de support pour 230
fichier de contenu (HTML), application outils XMLC 482–484
fichier temporaire dans un répertoire temporaire, création 75
fichiers
accès dans des environnements distribués 363
date de dernière modification/accès 54
arrondi des dates 55
contenu dynamique 55
en-têtes HTTP 107
modèles Tea, nommage 401
réception à l'aide des flots d'entrée de la requête 111–121
sérialisés, envoi 306
service 96–98
ViewFiles (exemple) 96
type MIME, obtention 95

fichiers de propriétés
applications web, correspondance avec les préfixes d'URI 25
astuce pour s'en souvenir 105
chaîne d'interrogation codée comme URL, création à partir de 302
conversion pour les redirections, API Servlet 2.3 560
ECS
filtres, configuration 459
installation 456
externes, connexions à une base de données basées sur 254
modèles WebMacro 434
TeaServlet
application NameApp, entrée 416
fichier exemple 407–408
fonctions de l'application, accès d'un modèle aux 410
Java contre 407
personnalisation du chemin 402
ToolApp (exemple) 418
URI
bibliothèques de balises 513
contextes de servlet 350
WebMacro
directives, liste dans 439
installation 433–435
outils, liste dans 436–438
timeout, spécification pour le contrôle des estampilles 446
file-extension-to-mime-type, correspondances 95
FileInputStream, classe, garantie de la fermeture 150
FileLocation, servlet (exemple) 94
FileNotFoundException, classe 132
FileOutputStream, classe 76
Filter, classe 549
FilterChain, classe 549
FilterConfig, interface 549
FilteredImageSource, classe 180
<filter-mapping>, élément 551
filtrage de servlets 13
filtres
API Servlet 2.3 548–551
déclaration dans le descripteur de déploiement 550
filtre de journalisation de la durée des requête 550
fonctionnalités de 549
ECS, utilisation pour les éléments 459
WebMacro 452
personnalisés 453
fin des données, écriture pour un format de compression 184, 186
finger, programme, exécution à partir des servlets 530–533
firewalls
connexions applet/serveur et 289
connexions non-HTTP, fonctionnalités bidirectionnelles inexistante 293
RMI, fonctionnement au travers 292
sockets brutes, interdiction 290
flots d'entrée
mise en cache des connexions faites avec un objet URL 302
objets URLConection 303
requête 110–121
classe InputStream 110
classe ServletInputStream 110
gestion des requêtes POST 111
réception de fichiers avec 111–121
type de contenu 111
vérification du type de contenu et longueur des données 111
requêtes de base de données retournant de longues chaînes de texte et binaires 284
flots de sortie
classe ServletOutputStream 125
codage de caractères, inconnu ou non supporté 125
obtention pour les messages e-mail 523
performances d'une servlet, amélioration avec 546
flush() 125, 431
flushBuffer() 129
fonctionnalités bidirectionnelles, connexions par socket non-HTTP 293
fonctionnalités de formatage, Java 377
fonctions, Tea
application outils, affichage des outils disponibles 420
applications installées, appel par des modèles Tea 410
gestion des requêtes/réponses 412
traitement du contenu 412
traitement du texte 410
#foreach, directive WebMacro 440, 443
itération sur la liste des outils 448
foreach, instruction, itération sur un ensemble d'objets 405
<form-error-page>, élément 633
<form-login-config>, élément 228, 634
<form-login-page>, élément 634
formulaires HTML 21
authentification basée sur 227–230
avantages et limitations 230
conversion de l'authentification de base en 227
pages d'ouverture de session 228

personnalisée 233–237
valeur FORM, balise <auth-method> 224
champs cachés dans 193–195
champs cachés, suivi de session avec 193–195
clients utilisant, connexion à des serveurs 248
contenu localisé, gestion 394–398
déchargement de fichiers, dans les requêtes POST 112
envoi de données par e-mail avec les servlets 524–525
frontal pour ResultSetServlet (exemple) 467
information de chemin supplémentaire, codage dans le paramètre ACTION 92
outils WebMacro pour 437
servlet Guestbook (exemple) 281
formulaires WML 161
forName() (Class) 253, 258
forward() 352, 506
Frame, classe 168, 174
français, langue, formats localisés 412
FreeMarker, moteur de templates 429
fuseau horaire, définition 376

G

Gefion Software LiteWebServer 8
GemStone/J Application Server 8
generateKey() 92
generateSessionId() 198
GenericServlet, classe 16, 71–73, 564–567
objet de configuration, passage à 43
support de la méthode sans argument init() 43
gestion de la terminaison d'une requête 24
gestion des exceptions 139–151
avec des serveurs 145
avec les servlets 145
classe ServletException 146
configuration des pages d'exception 147
Java, dans les servlets 11
page d'erreur générique dynamique 148
pages JSP 497
gestionnaires de sécurité
applets, s'exécutant sous 288
protection du serveur contre des servlets 12
servlets en tant que clients RMI, attribution à 536
GET, méthode 14, 15
appel de getNextMessage() contre 333
cache côté client et 54
clients de discussion, réception des messages 328
gestion par des servlets HTTP 17
HttpMessage, envoi 302
paramètres 87
réception de la requête WAP comme 164
restriction de l'accès aux servlets avec la balise <auth-constraint> 224
serveur SocketWatch, informations sur 540
servlet Hello World (exemple) 19
synchronisation 37
getAsciiStream() 284
getAttribute() 73
interface CORServletRequest 121
interface HttpSession 204
interface ServletContext 348
getAttributeNames()
interface HttpSession 204
interface ServletContext 74, 348
interface ServletRequest 121
getAuthType() 85, 225, 559
getAutoCommit() 268
getBinaryStream() 284
getBufferSize() 128
getBundle() 385, 388
getBundleNoFallback() 388
getByName() 82
getBytes() 518
getCodeBase() 299, 316
getColumnCount() 260
getColumnLabel() 260
getConnection() 253
getContentLength() 111
getContentType() 111, 114
getContext() 100, 431
distribution en dehors du contexte courant 351
getContextPath() 95, 560
getContextType() 413
getCookies() 199
getDate() 317, 324
getDateHeader() 108
getDateUsingHttpObject() 294, 305
getDateUsingHttpText() 294, 298
getDateUsingRMIObject() 294, 325
getDateUsingSocketObject() 294, 314
getDateUsingSocketText() 294, 314
getDefaultToolkit() 172
getElementLocale() 478
getFile() 114
getFileNames() 114
getFilesystemName() 114
getFilterConfig() 549
getFloat() 283
getGraphics() 168, 172
getHeader() 108
getHeaderNames() 109

getHost() 316
getId() 211
getImage() 168, 174
getInitParameter() 44, 71, 78
getInitParameterNames() 72, 78
getInputStream() 87, 110
getIntHeader() 108
getIntParameter() 518
getLastModified() 54, 55, 107, 182, 452
  cache côté serveur, utilisation dans 56
  classe PrimeSearcher (exemple) 55
  Guestbook, servlet (exemple), utilisation dans 282
  interface HttpServletRequest 56
getLocale() 393
getLocales() 393
getLog() 414
getMajorVersion() 80
getMethod() 24, 106
getMimeType() 95, 96, 165
getMinorVersion() 80
getName() 416
  classe HelloBean (exemple) 504
  classe HttpSessionBindingEvent 216, 552
  classe ServletContextAttributeEvent 552
  interface ApplicationConfig 414
  interface Principal 225
  page JSP, utilisation dans 494
getNamedDispatcher() 352, 354
getNextException() 258
getNextMessage() 329, 333, 337
  doGet(), enveloppe autour 333
getObject() 257, 260, 261
getObject().toString() 260
getOutputStream() 125, 379
getParam() 441
getParameter() 22, 87, 111, 114, 518
getParameterMap() 559
getParameterNames() 88, 113
getParameterValues() 87, 88, 114, 518
getPathInfo() 93
getPathTranslated() 93, 96, 98
getProperties() 414
getProperty() (système) 543
getProtocol() 106
getQueryString() 88
getReader() 87, 110
getRealPath() 94
getRegistryHost() 325
getRegistryName() 323, 325
getRegistryPort() 323, 325
getRemoteAddr() 82
getRemoteHost() 82, 84
getRemoteUser() 85, 86, 107, 192, 225
getRequest() 432
getRequestDispatcher() 351
getRequestURI() 105
getRequestURL() 104, 559
getResource() 100, 172, 174
  classe ServletUtils 102, 103
  interface ServletContext 98
getResourcePaths() 559
getResponse() 432
getResultSet() 262, 263
getRootCause() 146
getScheme() 106
getServerInfo() 73
getServerName() 73
getServerPort() 69, 73
getServlet(), désapprouvée dans l'API 2.1 347
getServletContext() 348, 551
getServletContextName() 559
getServletInfo() 22
getServletName() 72
getServletPath() 105
getSession() 204, 552
getSocketPort() 313, 334
getStackTraceAsString() 144
getString() 257, 386
getStringParameter() 519
getTemplate() 431
getTime() 55
getTools() 452, 509, 511
getUnavailableSeconds() 147
getUpdateCount() 262, 263
getUserPrincipal() 225
getValue() 552
getWebContext() 431
getWriter() 19, 125
  setLocale(), appel avant 393
getXXX(), méthodes, interface CallableStatement 283
GIF, format d'image 168
  algorithmes de compression LZ et LZW 169
  codeur Acme, téléchargement 168
  lecture à partir de la base de données 284
glyphes, affichage dans un navigateur 379
GNU, General Public License (GPL) 525
graphique, compteur, combinaison d'images pour créer un 175–178
graphiques
  contexte hors écran, création d'images dans 168
  format d'image Wireless Bitmap (WBMP) 157
  génération dynamique 169–171
  (voir aussi images)
GrayscaleImageFilter, classe 180
grep, programme 533

guillemets
caractères de contrôle SQL et chaînes de bases de données entre guillemets 264
simples ou doubles, chaînes dans des blocs de code Tea 405
simples ou doubles, dans le fichier de propriétés de TeaServlet 408
valeurs d'attributs (directives JSP), utilisation avec 495
valeurs d'attributs XML, entourer 27
gzip (ou x-gzip), schéma de codage 183
GZIP, format de compression 183
GZIPOutputStream, classe 185

H

handle() (exemple ToolServlet) 451
handleClient() 313, 333
handleForm() 281
handleMalformedCount() 517
handleNoCount() 517
HandlerException, classe 452
HashMap, classe 545
Hashtable, classe, synchronisation 545
HEAD, méthode 16
servlet Hello (exemple) 23
HeaderSnoop (exemple) 109
Hello, classe, code source autogénéré avec XMLC 472–476
Hello, modèle (WebMacro) 430
hello1.jsp (exemple) 491
servlet d'arrière-plan auto-générée 492–494
hello2.jsp (exemple), utilisant une déclaration JSP 494
HelloBean, classe (exemple) 503
hello.html, maquette de page web, compilation en classe avec XMLC 470
HelloManipulation, classe 476
Help, servlet, invocation avec une URL incluant l'identifiant de session 198
heures
valeur long, représentation avec 55
HolisticCounter (exemple) 38
<home>, élément 634
hôtes
serveur de registre RMI, obtention pour 325
sessions, liaison à un hôte particulier 364
HTML
<A HREF>, liens sur une page web, trouver et lister 526–529
code de servlet, insertion directe dans (JSP) 490
contenu de modèle, utilisant dans 430
DaytimeApplet, fichier référençant 296
documents, compilation en classes Java 470–476
échappement des caractères spéciaux, filtre WebMacro pour 453
entités caractères 372, 659–664
dans les données d'un ensemble de résultats d'ECS 459
fichier de contenu, application outils XMLC 482–484
fichier de modèle, application outils XMLC 480–482
génération orientée objets 455–468
composants de page comme des objets 455–457
ensembles de résultats, affichage 457–468
outils de conception 471
WML contre 157
XML, différences 27
HTML, formulaires
(voir formulaires HTML)
HTMLDocument, interface 475
HTMLElement, objet DOM 476
htmlKona, produit de WebLogic 455
HTMLObjectImpl, classe 475
HtmlResultSet, classe (exemple) 259
HtmlSQLResult, classe (exemple) 262
affichage des résultats, servlet d'annuaire téléphonique 265
HTTP
API Servlet, référence 593–620
authentification 220–226
digest 221
retrouver les informations sur 225
clients de discussion
notification de nouveaux messages 333
poster et lire des messages 328
codes d'état 651–657
communications applet/serveur
à base d'objets 304–307
à base de texte 297–304
combinaison avec non-HTTP et RMI 293
connexion d'applet à un programme CGI sur le serveur 288
contenu compressé, schémas de codage 183
couche de transport RMI, opérer dans 292
en-têtes 106–110
cookies, envoi avec 198
réponse 132–139
requête, liste de 107
(voir aussi en-têtes)
fonctions de restriction des accès du serveur 84
information de chemin supplémentaire 92–95
raisons de l'existance 93
les bases 13–16

protocole sans état, problèmes avec 191
requêtes, réponses et en-têtes 14
(voir aussi en-têtes ; requêtes ; réponses)
servlet de registre RMI, accès via 323
spécifications des normes, cookies et 198
HTTP, méthodes
autres 16
GET et POST 15
(voir aussi GET, méthode ; POST, méthode)
http, schéma de requête (requête non sûres) 106
HttpChatApplet (exemple) 334–337
HttpClient, classe (exemple) 541–543
HttpMessage, classe 299–304, 529
envoi d'objet ou de fichier sérialisés 306
envoi de requête POST 303
localisation pour utilisation dans une applet 303
<http-method>, élément 635
HTTPS, protocole 240
attributs de requête, API Servlet 2.3 558
communications applet/serveur sûres 289
communications sûres à partir d'une applet 303
(voir aussi SSL)
https, schéma de requête 106
HttpServlet, classe 16, 598–601
classe RemoteHttpServlet 322
HttpServletRequest, interface 18, 19, 56, 85, 601–606
accès aux valeurs des en-têtes 108
cache côté serveur et 56
classe MultipartRequest et 121
cookies, travail avec 199
types d'authentification, changements dans l'API Servlet 2.3 559
HttpServletRequestWrapper, classe 550
HttpServletResponse, interface 18, 19, 606–611
codes d'état, définition 130
cookies, ajout à 199
HttpServletResponseWrapper, classe 550
HttpSession, interface 203, 611–615
messages en attente dans 333
HttpSessionActivationListener, interface 553
HttpSessionAttributesListener, interface 552
HttpSessionBindingEvent, classe 216, 615
HttpSessionBindingListener, interface 215, 616
connexions à une base de données, travail avec 277
HttpSessionContext, interface 617
désapprobation 211
HttpSessionListener, interface 552
HttpsMessage, classe 303
HttpURLConnection, classe 211
HttpUtils, classe 104, 618–620
désapprobation, API Servlet 2.3 559

I

I18N (voir internationalisation)
IBM AlphaWorks, outil de débogage DAT 543
IBM WebSphere Application Server 8
identifiants
attribut id pour les beans 501
attributs ID, marquage des éléments HTML pour DOM 476
éléments <ID>, retour comme des objets HTMLElement 478
numériques, venant des autorités de certification 239
identifiants de session 211–213
codage dans XMLC 488
détection s'ils viennent d'un cookie ou d'une URL codée 213
génération 198
réécriture d'URL pour l'inclusion 212
réécriture des URL pour inclure 196
servlet écrivant un lien contenant 212
suivi de sessions avec 195
validité, déterminaison 213
#if, directive WebMacro 439, 443
if/elseif/else, construction, gestion dans WebMacro 443
If-Last-Modified, en-tête 107
If-Modified-Since, en-tête 54, 55
IIOP (Internet Inter-ORB Protocol), CORBA 292
IllegalArgumentException, classe 108, 364
IllegalStateException, classe 110, 125, 511
charsets inconnus, lancement pour 379
reset() (Servlet Response), lançant 129
sessions invalides 204, 211
image Bonjour tout le monde, génération 167
images 165–183
classe Image 168
composition 172–178
combinaison d'images 175–178
dessin par dessus d'une image 172–175
effets spéciaux 178–183
classe ImageProducer 180
conversion en niveaux de gris 179–181
mise en cache des images converties 181–183
fichier GIF, lecture à partir de la base de données 284
format Wireless Bitmap (WBMP) 157
génération 166–172
« Bonjour tout le monde ! » 167
classe BufferedImage 168
contexte graphique hors écran, utilisation 168
formats de fichier, codage dans 169

graphique généré dynamiquement 169–171
modèles Tea, création avec 400
requêtes de renvoi, problèmes avec les chemins d'URI relatifs 355
importation de classes et de paquetages dans les JSP 496
importNode() 487
include() 352, 355, 505
#include, directive WebMacro 440
indices, objets ResultSet 260
InetAddress, classe 82
information de chemin
attributs
requêtes d'inclusion, retrouver pour lex composants 359
chemin de classes du serveur 40
copie des fichiers .class pour les objets partagés 350
inclusion des classes stub/squelette 319
chemin de classes, recherche de rmic 318
classes utilitaires globales, installation dans le chemin de classes de la servlet 40
composants EJB, noms de recherche JNDI 368
contexte, obtention 95
cookies, spécification pour 200
débogage de servlets 543
distribution de requêtes
chemins d'URI vers des composants 351
URL 351
fichier TeaServlet.properties, personnalisation 402
fichiers inclus, JSP 505
FileLocation (exemple) 94
racine du contexte, relative à 351
racine du document 94
relative, dans la distribution de requête 351
requêtes de renvoi 354
requêtes de servlet HTTP 93–98
ressources dans le contexte de servlet, obtention 559
supplémentaire, dans la réécriture d'URL 92, 196
supplémentaire, requêtes HTTP 92–95
conversion en chemin sur le système de fichiers 93
formulaires HTML, codage dans 92
programmes CGI 93
type MIME, obtention 95
ViewFile (exemple), retour des fichiers avec 96
système de fichiers plein 92
TemplatePath, inclusion d'un fichier cible #parse 440
URL relatives pour les fichiers d'image, problèmes avec les requêtes de renvoi 355
information HTML, envoi 123–153
avec des connexions persistantes 126
codes d'état 129–132
en-têtes HTTP 132–139
erreurs et exceptions 139–151
mise en tampon des réponses 127–129
servlet KeepAlive (exemple) 127
réponses normales 124
structure des réponses 123
informations de l'environnement, obtention 69–121
CGI, servlets contre 69
clients 82–121
accueil personnalisé, création 86
attributs supplémentaires 121
clé de licence, génération 90–92
déterminer ce qui a été demandé 104
en-têtes de requêtes 106–110
flot d'entrée 110–121
information de chemin 92–95
lecture à partir d'une ressource abstraite 98–100
machine du client 82
requêtes 87–90
restriction de l'accès 83
servir des fichiers 96–98
servir des ressources 100–104
utilisateurs 84–85
noms des servlets 72
paramètres d'initialisation des servlets 71, 72
serveurs 73–82
types MIME 95
variables d'environnement CGI et méthodes des servlets 70
init() 40–47
classe PrimeSearcher (exemple) 52
classe ToolApp 418
création du Frame dans 174
gestion des exceptions 146
interface Application 413
réserve de connexions à une base de données 270
sans argument, classe GenericServlet supportant 43
servlet Guestbook (exemple) 281
servlets d'arrière-plan, JSP 495
servlets, écoute des applets 308
surcharge de la version prenant un ServletConfig 544
ToolViewServlet (exemple) 487
InitCounter (exemple) 42–44

InitDestroyCounter (exemple) 44–47
  sauvegarde de l'état tous les 10 accès 47
initialisation
  informations du contexte, obtention 78
  servlets 41
    lancement d'une UnavailableException temporaire 147
  threads 313
  (voir aussi paramètres d'initialisation ; init())
initialisation des contextes de servlet et des sessions 551
initialisation, paramètres
  obtention pour les servlets 71
<init-param>, élément 635
  placements multiples dans <servlet> 42
Inprise Application Server 4 8
INSERT, instruction SQL 262
  servlet Guestbook (exemple), utilisation dans 281
installation
  bibliothèques de balises personnalisées dans les JSP 512
  TeaServlet 401
  WebMacro 433–438
    fichier de propriétés 433–435
    outils du contexte 436–438
  XMLC 470
instances (servlet), comptage 38
instances d'objets
  persistance d'une servlet comme 34
  verrouillage 37
instances en réserve pour les noms de servlets 48
instantiate() 501
instructions
  SQL (voir SQL)
  Tea, Java contre 405
  WebMacro (voir directives ; WebMacro)
instructions préparées, SQL 264
  réutilisation 266
  servlet Guestbook (exemple), utilisation dans 281
intégration des servlets avec le serveur 12
intégrité des données
  cryptographie à clés publiques/privées, garantir avec 238
  garantir la non-modification pendant le transit 242
interception d'exceptions 145
internationalisation 371–398
  API Servlet 2.3, sélection du codage de caractères 553
  charsets
    écriture d'une sortie codée 378–381
    liste 665–666
    navigateurs, utilisation de 377
  classe ParameterParser, spécification du charset avec 518
  formulaires HTML avec du contenu localisé 394–398
  langues multiples, sortie sur la page 381–384
    UTF-8, écriture 382–384
  langues non occidentales, sortie d'une page dans 377–381
  langues occidentales, sortie des pages dans 372–375
    échappements Unicode 374
    entités caractères HTML 372
  locales fournies par le système 393
  localisation des formats 375–377
    dates 375
  négociation dynamique de la langue 384–393
    lots de ressources 385–388
    préférences de charset 385
Internet Explorer (voir Microsoft ; Internet Explorer)
Internet Information Server (IIS), support des ASP 5
interpréteurs SQL 250
introspection 500
  beans embarqués dans les JSP, définition des propriétés via 502
  classes d'objet partagé 350
  interface d'auditeur implémentée, détermination 552
  WebMacro, utilisation 433, 436
Introspector, classe 436
invalidation de session 206
  authentification à base de formulaire 230
  incapacité du serveur à détecter 206
  manuelle (exemple) 210
invocation
  directe de méthodes, WebMacro 443
  méthodes distantes (voir RMI)
  servlets de type WebMacro 435
IOException, classe 145, 150
  lancement quand tampons pleins, JSP 496
  trace de la pile, fournit pour 144
IP, adresse 191
iPlanet/Netscape
  API d'extension du serveur WAI 4
  Application Server 8
  server-side JavaScript (SSJS) 5
  Web Server Enterprise Edition 7
  (voir aussi Netscape Navigator)
isa, mot-clé (langage Tea) 405
ISAPI (API d'extension du serveur) 4
isCommitted() (ServletResponse) 128
isErrorPage, attribut JSP 496, 497

isNewWithin() 448
ISO-8859-1 (Latin-1), caractères 125
isPermanent() 147
isProperty() 443
isRequestedSessionIdFromCookie() 213
isRequestedSessionIdFromURL() 213
isRequestedSessionIdValid() 213
isSecure() 243
isUpdatedWithin() 448
isUserInRole() 225
  API Servlet 2.3, clarifications des rôles 560
<iterate>, balise (Apache Struts) 512
iterator() 448

J

J Application Server 8
japonais, langue
  charset pour 377
  formulaire codé Shift_JIS, lecture des paramètres 554
  servlet HelloJapan (exemple) 378
  servlet HelloJapanReader (exemple) 380
JAR (Java Archive), fichiers
  fichier struts.jar, installation 512
JAR (Java Archive), fichiers 25
  dépendances, expression dans les fichiers WAR 554
  ECS, installation dans le chemin de classes du serveur 456
  WebMacro, installation 433–435
  XMLC, installation 477, 487
Java
  code dans les pages JSP, raisons de l'éviter 499
  expressions régulières, utilisation 525–529
  fichiers de propriétés 254
  fonctionnalités de formatage 377
  servlets 5
  types de données
    primitifs, affichage 125
    sûreté du typage fort 11
  version, trouver 82
Java 2 Platform, Standard Edition 1.2 (J2SE 1.2 ou JDK 1.2) 548
Java 2, Enterprise Edition (J2EE)
  applications Java, support de 8
  Enterprise JavaBeans (EJB) 11, 362
  intégration des servlets avec 365–370
    composants EJB, références à 368
    distribution des servlets 370
    entrées de l'environnement 366–368
    fabriques de ressources externes, références à 369
    rôles, développement d'applications 365
    JDBC 2.0, base de données et pilote, support requis 286
    JSP comme composant fondamental 489
    mécanisme de sécurite, API Servlet 2.2 221
    servlets comme composant fondamental ix
    servlets d'entreprise 361–370
    Version 1.3 (J2EE 1.3) 548
Java Apache Project
  ECS (Element Construction Set) 455
  module JServ 9
Java API for XML Parsing (JAXP) 365
Java Database Connectivity API (voir JDBC)
Java Image Management Interface (JIMI), outil 169
Java Messaging Service (JMS) 365
Java Naming and Directory Interface (voir JNDI)
Java Native Interface (JNI) 534
Java Plug-In
  limites du suivi de session, applets s'exécutant dans 303
  mise à jour de la JVM pour les navigateurs 288
Java Software (division de Sun Microsystems), définition des servlets 6
Java Transactions (JTA) 365
Java Virtual Machines (voir JVM)
Java Web Server 8
java.* et javax.* , noms d'attributs 121
java.* et javax.*, paquetages 74
java:comp/env, contexte 367
java.awt, paquetage 168
JavaBeans
  JSP, utilisation avec 500–504
    affichage de « Bonjour » 503–504
    contrôle des propriétés d'un bean 502
    embarquement dans une page JSP 500–502
    page toolview.jsp (exemple) 509–511
  portée, gestion par le serveur 500
javac, compilateur 20
java.io.File, objet, obtention 114
JavaServer Pages (voir JSP)
java.sql, paquetage JDBC 2.0 286
java.util.zip, paquetage 183
javax.*, paquetages 121
javax.servlet et javax.servlet.http, paquetages
  servlets HTTP, utilisation 16–18
  sortie en version en open source ix
javax.servlet, paquetage 563–591
javax.servlet.context.tempdir, attribut 75
javax.servlet.http, paquetage 593–620
javax.servlet.jsp, paquetage 491
javax.sql, paquetage, support des pilotes et des bases de données JDBC 2.0 286

JClass Chart, paquetage 172
JDBC (Java Database Connectivity) 247, 251–264
  abstraction des détails avec establishConnection() 71
  ANSI SQL-2 Entry Level (JDK 1.1) 250
  champs null, gestion 261
  communications applet/serveur, utilisation dans 291
  connexions, obtention à partir des servlets 255
  ensembles de résultats SQL 259–261
  exceptions SQL, gestion 258
  instructions préparées, utilisation 264
  interaction du programme avec des bases de données 251
  mise à jour de la base de données 262
  pilotes 251
    catégories 252
    connexions à une base de données, obtention 253–255
    identifiants d'URL, inclusion dans 253
    liste, site web 253
    Sybase 265
  requêtes SQL, exécution 256–258
    servlet DBPhoneLookup (exemple) 257–258
  techniques avancées 282–285
    grands types de données, gestion 284–285
    procédures stockées 282
  transactions
    ordre de traitement dans une servlet (exemple) 268
    validation/annulation 268
    (voir aussi transactions)
  URL, bases de données 253
  Version 2.0, fonctionnalités supplémentaires 285
JDBC-ODBC, pont 252
  chargement du pilote de base de données avec 253
  format d'URL 253
  risques potentiels 252
JDK 1.2 548
JDK, version, déterminaison avec la classe VersionDetector 80
JDOM, API, manipulation de XML avec 420
JDOM, modèle d'objet, support futur par XMLC 480
jeu de carte, métaphore, WML 157–158
jeux de caractères
  déclaration dans les descripteurs de déploiement 28
  Unicode, dans une sortie multilingue 381
Jigsaw Server (W3C) 8
JMS (Java Messaging Service) 365
JNDI (Java Naming and Directory Interface) 365, 366
  entrées de l'environnement, localisation de ressources externes avec 366–368
  fabriques de ressources externes, noms de recherche 369
  serveur, placement de connexions aux bases de données dans 255
  support de la recherche, connexions aux bases de données 286
JNI (Java Native Interface) 534
jointure de tables de bases de données 250
journalisation 46
journalisation des erreurs dans un fichier 143
journaux
  application, obtention 414
  contrôle dans le débogage de servlets 536
journaux d'erreur (serveur), vérification dans le débogage des servlets 536
journaux d'erreur, vérification dans le débogage des servlets 536
JPEG, format d'image 169
JProbe (Sitraka), outil de profilage 546
JRun Server 9
JRun, conteneur de servlets ix
  fonctionnalités 9
JServ, module 9
  liaison de sessions à 364
JSP (JavaServer Pages) 5, 152, 489–516
  actions 500
  application outils 506–511
    fichier d'en-tête 507
    fichier de base de page 508
  bibliothèques de balises personnalisées 511–516
    application outil utilisant 513–516
    liste 511
  code de servlet, insertion directe dans un fichier HTML statique 490
  code Java dans, raisons de l'éviter 499
  directives 495–500
    pages d'erreur, utilisation 497–499
  expressions et déclarations 494
  hello1.jsp (exemple) 491
  inclusions et renvois 504
  JavaBeans, utilisation avec 500–504
    affichage de « Bonjour » 503
    contrôle des propriétés d'un bean 502
    embarquement de beans 500–502
  logique de présentation, utilisation pour 490
  page d'accueil, site web 490

précompilation 409, 494
scriptlets, problèmes avec 499
servlets d'arrière-plan 492–494
<jsp:forward>, action 506
<jsp:include>, action 505
<jsp:param>, balises
dans une action <jsp:forward> 506
dans une action <jsp:include> 505
<jsp:setProperty>, action 502, 509
<jsp:useBean>, action
attributs 500
corps, interprétation 501
page toolview.jsp, utilisation dans 509
toolview-tag.jsp (exemple), utilisation dans 515
jspDestroy() 495
<jsp-file>, élément 636
jspInit() 495
JTA (Java Transactions) 365
JVM (Java Virtual Machines)
classe du pilote JDBC, chargement 253
exécution de toutes les servlets dans une même 33
mises à jour pour les navigateurs avec le Java Plug-In 288
problèmes de chemin de classes, débogage 543
registre RMI, utilisation pour 323
servlets, s'exécutant sur 5

K

Kannel (passerelle WAP à sources ouvertes) 156
KavaChart, paquetage 172
-keep, option XMLC 472
KeepAlive, servlet (exemple) 127
Kettle, environnement auteur graphique (Tea) 428
key (paramètre d'initialisation), verrouillage d'une servlet au serveur 76
KeyedServerLock (exemple) 76–78
déverrouillage 90–92
keyFitsServer() 78
KL Group, paquetage JClass Chart 172

L

L10N (voir localisation)
lancement d'exceptions 145
langages de balisage
WML (Wireless Markup Language) 155–164
(voir aussi HTML ; WML ; XML)
langages de programmation
bases de données, procédures stockées
Oracle PL/SQL 282
préprocesseur Perl et C, utilisation avec WebMacro 436
procéduraux contre déclaratifs 250
SQL (voir SQL)
Tea 400
langages de script
CGI 2–4
FastCGI 3
mod_perl, utilisation avec 3
Perl, écriture en 2
PerlEx, utilisation avec 3
JavaScript (pour les scripts côté serveur) 5
spécification pour les sections de code JSP 497
WMLScript 157
langues
charsets pour 665–666
codes 373
multiple, sortie avec le codage UTF-8 382–384
négociation dynamique 384–393
lots de ressources 385–388
préférences du client 384
occidentales, charset pour 125
<large-icon>, élément 636
Last-Modified, en-tête 54, 55, 107
Latin-1, charset 377
accélération de la sortie dans WebMacro 431
entités caractères 659–664
liaison
objets au registre RMI 322
objets aux sessions 204, 215–217
renvoi des noms de 204
objets sous des noms données 348
licence
de type Apache
Tea 399
WebMacro 429
Enhydra Public License (EPL) 469
GNU General Public License (GPL) 525
liens (<A HREF>), trouver et lister 526–529
ligne d'état, réponse HTTP 14
lignes, navigation dans les ensembles de résultats 256
liste des fichiers d'accueil, ajout d'index.wml à 160
LiteWebServer (Gefion) 8
<load-on-startup>, élément 52, 636
loadTools() 418, 451
LocaleNegotiator, classe (exemple) 386, 388–393
locales 375
fournies par le système 393
locales, variables 36
d'instance, changement en 38
Tea, traitement des 405

LocaleToCharsetMap, classe 388
localhost, nom réel du serveur contre 78
localisation 372, 375–377
  date et pourcentage, français 412
  filtrage des dates, WebMacro 453
  formats de date 375
  formulaires HTML avec un contenu local 394–398
  lots de ressources, utilisation 385–388
<location>, élément 637
  cibles dynamiques, lancement d'exceptions 148
  spécification des pages d'erreur 142
log() 143
<login-config>, élément 637
  authentification de base, passage à celle basée sur un formulaire 227
  ordre dans le fichier web.xml 223
  spécification de la méthode d'ouverture de session 224
logique applicative, inclusion dans des pages JSP 512
logique de présentation
  JSP, utilisation 490
  séparation du contenu 151
  vue (architecture MVC) 151
logique métier
  classes gérant les requêtes de client RMI, HTTP et par socket 293
  Enterprise JavaBeans, implémentation avec 362
  environnement WebMacro, programmation 430
  inclusion dans le tiers intermédiaire 248
long, type de données, stockage des heures 55
longueur des données, flots d'entrée des requêtes 111
lookup(), registre contre nommage 325
lots de ressources 385–388
  anglais et japonais (exemples) 386, 395
  localisation avec la classe LocaleNegotiator 388–393
Lutris, Enhydra Application Server 8
LZ et LZW, algorithmes de compression 169

M

MacroView, servlet (exemple) 444–446
MailMessage, classe 523
  pièces jointes, non supportées 525
main(), méthode, absence dans les servlets 16
manipulation, classes, XMLC 469, 476–480
  maquette de liste de langues 478–480
  page Hello 476
ManualInvalidate, servlet (exemple) 210
MathTool, classe (exemple) 437
  opérateurs de comparaison, utilisation 439
MD5, algorithme de chiffrement 221
MediaTracker, classe 174
mémorisation d'URL 15
messages d'accueil, personalisation pour les clients 86
messages électroniques, envoi avec les servlets 522–525
  classe MailMessage 523
  données de formulaire 524–525
MessageSink, classe 333
MessageSource, classe 333
métaphore du jeu de cartes, WML 157–158
méthodes
  chaînage 457
  cycle de vie d'une session, gestion 209
  définition de nouvelles, déclarations JSP 494
  ensembles de résultats, retrouver les données 260
  HTTP 14
    DELETE 16
    GET et POST 15
    HEAD 16
    OPTIONS 16
    TRACE 16
    (voir aussi GET, méthode ; POST, méthode)
  invocation directe avec WebMacro 443
  invocation, remplacement par RMI 292
  natives, utilisation dans les servlets 534
  objet, invocation directe dans WebMacro 436
  primitives de Java, écriture 125
  servlets HTTP, correspondance avec les variables d'environnement CGI 70
  synchronisées 36
  Value, modification des noms 204
méthodes get et set pour les cookies 199
-methods, option XMLC 476
Microsoft
  Active Server Pages (ASP) 5
  Internet Explorer
    cookies, support de 198
    déchargement de fichiers, support 112
    fonctionnalités d'internationaliation 379
    ignorer le type de contenu attribué par le serveur 104
    préférences de la langue, spécification 384
    push du serveur, non supporté 186
  ISAPI (API d'extension du serveur) 4
middleware 248
migration de session 364
  auditeur pour la gestion 553
<mime-mapping>, élément 159, 637

<mime-type>, élément 638
mini carte des boissons, WML (drinks.xml) 157
mise à jour de l'état, avantages du push du serveur 186
mise à jour de la base de données 262
  avec SQL 251
  date des dernières, obtention 452
  optimisation du traitement des transactions 270
mise à jour des pages
  faite par le client 186
  faites par le serveur 186
mise en cache
  connexions faites avec un objet URL 302
  données d'un ensemble de résultats, JDBC 2.0 286
  images 181
  informations (non sauvées), écriture pendant destroy() 42
  informations de la base de données 209, 254
  mises à jour de base de données, modifications après 282
  navigateurs, de la sortie des requêtes précédentes 544
  sortie de modèle (WebMacro) 452
mise en cache côté client 54–56
mise en cache côté serveur 56–67
  Guestbook (exemple), utilisation de CacheHttpServlet 56–59
mise en tampon
  entrée et sortie, réglage des performances des servlets 545
  journaux du serveur, arrêt pour le débogage d'une servlet 537
  réponses 124, 127–129
    API Servlet 2.3, changements dans 559
    contrôle 128
    définition de la taille du tampon 127
    effacement du tampon 128
    JSP 496
    requêtes de renvoi et 354
    servlet Buffering (exemple) 129
    servlet KeepAlive (exemple) 127
    sortie sur une requête client arrêtée 150
    vidage des tampons 129
mod_perl, module, utilisation avec le serveur Apache 3
modèle 151
modèle pull/push, WebMacro 452
modèles
  fichier HTML, application outils XMLC 480–482
  sortie d'un composant ECS, inclusion 459
  Tea 399–428
    environnement de création Kettle 428
    fonctions d'une application installée, appel 410
    informations sur la requête, obtention (exemple) 404
    précompilation pour la distribution 408
    rechargement de fichiers 408
    régions de code et de texte 401
    toolview1 (exemple) 423
    toolview2 (exemple), ajout de la décoration de la page 423
  WebMacro 436, 442–447
    blocs de texte, analyse avec un autre analyseur 441
    directive #param, spécification de valeurs pour la servlet 441
    outils du contexte et directives disponibles 442
    recherche par une servlet 431
    servlet utilisant 430
    stockage, chemin de répertoire pour 434
    traitement 446
Modèle-Vue-Contrôleur (MVC), architecture 151
  architecture de type Model 2, JSP 511
  contrôleur, ToolServlet (exemple) 513–515
  séparation du modèle et de la vue, Tea 401
modification/accès, date
  cache côté serveur, utilisation dans 56
  fichiers de modèle Tea 408
  mises à jour de la base de données, configuration 282
  modèles WebMacro, fréquence des contrôles 446
  obtention de la dernière 452
  rechargement de classes, contrôle pour 40
  servlet de base de données Guestbook 282
  (voir aussi getLastModified())
moment de la conversion, création de la servlet d'arrière-plan 505
moment de la requête, expression d'attribut au 503
moment de la requête, inclusion au 505
Mon Yahoo!, suivi de session 206
moniteur (verrou), sur une instance d'objet 37
moteur de recherche
  frontal (exemple SearchView) 353
  serveur (exemple SearchLogic) 352
moteurs de template 152, 429
  en open source 429
motifs d'URL
  correspondances de servlet, configuration pour 29–31
  élément <url-pattern> 647

filtres, application à 551
instance de servlet et 38
spécification des URL à protéger 224
motifs de domaine, spécification pour les cookies 199
mots de passe
ajout à la requête avec HttpMessage 303
authentification à base de formulaire, risques 230
authentification digest 221
codage Base64, décodeur pour 231
codage Base64, limites de la sécurité du 220
dans l'ouverture de session d'un client 192
(voir aussi authentification)
multimédia, contenu
envoi 155–189
contenu compressé 183–186
push du serveur, utilisation 186–189
gestion dans la base de données d'un 284
images 165–183
multipart/form-data, type de contenu 112
déchargement et analyse 112–121
MultipartParser, classe 116
MultipartRequest, classe 112–121
code 116–121
MultipartResponse, classe 187–189
MVC (voir Modèle-Vue-Contrôleur (MVC), architecture)
MyConnectionManager, classe (exemple) 553

N

name, propriété, beans 504
NameApp, classe (application Tea) 413
NameContext, classe (exemple) 414
native, méthodes, utilisation dans les servlets 534
Native-Protocol All-Java Driver 253
nature déclarative de SQL 250
navigateurs
authentification digest, support de 221
champs de formulaire cachés, support 195
charsets, affichage des réponses avec 377
clés du serveur, contrôle de la validité 240
communication RMI, support limité 292
compression, support de 183
contrôle d'accès, applets partiellement de confiance 288
cookies
acceptation de 202
non acceptés, suivi de session et 212
utilisation de 198
débogage de servlet, utilisation dans 544
déchargement de fichier, support 112
fenêtres multiples, suivi de session et 213
fonctionnalités d'internationalisation 379
formulaires HTML avec du contenu localisé 394
Internet Explorer (voir Microsoft ; Internet Explorer)
Netscape Navigator (voir Netscape Navigator)
préférences pour la langue, spécification 384
rechargement de pages, en-tête If-Modified-Since 54
réécriture d'URL, support pour 198
requêtes d'applet, gestion 212
serveurs non-HTTP, impossibilité de communiquer avec 290
URL de requête et 104
WML, support pour 158
Netscape Navigator
cookies, support 198
déchargement de fichiers, support 112
fonctionnalités d'internationalisation 379
préférences de la langue, spécification 384
Netscape, spécification des cookies (site web) 198
next() 256
Nexus Web Server (conteneur embarquable) 9
NileBooks, servlet (exemple)
inclusion d'un élément de catalogue 356
inclusion de plusieurs éléments de catalogue 356–358
NileItem, servlet 358
niveaux de gris, conversion d'une image en couleurs en 179–181
NoClassDefFoundError 82, 350
nœuds, DOM 487
nom de classe complet (NCC)
entrées de l'environnement 367
<exception-type>, ServletException 148
fabrique de ressources, spécification dans la balise <res-type> 370
interface distante d'un EJB, spécification avec la balise <home> 369
nom de machine
machine de client distant, obtention 82
obtention pour une socket 316
nom de recherche pour taglib 512
nom du client, obtention (application Tea) 416
nombres
personnalisation de l'affichage avec ECS 461
premiers, recherche 50–52
noms
attributs 74, 348
cookies 198
des utilisateurs acceptés par le serveur, obtention 85

objets liés à une session, obtention 204
objets liés/déliés, sessions 216
paquetages, conventions 74
paramètres d'initialisation d'une servlet, obtention 72
serveurs 73
localhost contre nom réel 78
servlets, obtention 72
noms d'utilisateur
« anonymous » 233
ajout à la requête avec HttpMessage 303
authentification de base, utilisation dans 220
codés Base64, décodeur 231
suivi de la session du client avec 192
TCP/IP, accès facile aux 220
(voir aussi authentification ; mots de passe)
noms enregistrés, servlets 72
noms sur le système de fichiers, obtention 114
non locales, variables (servlet), threads client manipulant 36
non sûres, applets 287
non-HTTP, sockets (voir connexions par socket)
NoSuchFieldException, classe 82
Not Found (404), code d'état 124
NotBoundException, classe 325
notify() 333
NullPointerException, classe 146, 404, 559
NumberFormatException, classe 109, 518
numériques, certificats 237–239

O

O'Reilly WebSite Professional 8
ObjectInputStream, classe 306
ObjectOutputStream, classe 305
objets
collection, itération sur 405
équivalence, vérification 403
liaison aux sessions 215–217
liaison sous des noms d'attributs donnés 348
liés aux sessions, retour des noms 204
non nécessaires, éviter la création 544
propriétés
affichage avec Kettle 428
recherche par introspection, WebMacro 436
session, migration dans des environnements distribués 363
texte HTML traité comme 455–468
composants de page 455–457
ensemble de résultats, affichage 457–468
objets administrés, support dans J2EE 1.3 548
objets de base de données, réutilisation 265–267
connexions 265
instructions préparées 266
objets distants
auto-exportation 322
classes stub et squelette 292
objets sérialisés
classes non dans l'API Java Core, localisation pour les applets 306
communications applet/serveur, utilisation dans 291
Date, représentation de l'heure courante 304–306
envoi 306
types de contenu non représentant 305
Open Market FastCGI 3
openStream() 302
opérateurs booléens (&&, || et !), utilisation avec les directives #if 439
opérateurs de comparaison, utilisation par WebMacro 439
optimisation, traitement des transactions 269
OptimizeIt! (Intuitive Systems), outil de profilage 546
Optional Package (Javax.sql), support avancé des bases de données 286
options de compilation, XMLC 471
options de la ligne de commande
informations de profilage 546
XMLC 471
OPTIONS, méthode 16, 17
Oracle
Application Server 8
pilote JDBC-Thin 253
procédure stockée PL/SQL 282
SQL*PLUS, interpréteur SQL 250
ordre de recherche, correspondances d'URL dans les descripteurs de déploiement 30
Orion Application Server 8
out.close() 184, 186
outils de conception HTML 471
outils de création de contenu, liste 416–427
outils de profilage, ajustement des performances d'une servlet avec 546
outils du contexte, WebMacro 436–438
disponibles aux modèles, affichage 442
MathTool (exemple) 437
outils tiers pour le débogage de servlets 543
outils, application
JSP 506–511
bibliothèque de balises personnalisées, utilisation 513–516
WebMacro 447–452
XMLC 480–488
out.println(), limitations de 151

OutputOptions, classe 477
OutputStream, classe 110, 125
  contenu compressé 185
ouvertures de session
  authentification à base de formulaire, spécification de page 228
  client 192
  domaine, spécification 224
  servlet LoginHandler (exemple) 236

P

page, directive JSP 495
pages
  accueil du serveur web, obtention d'une référence à 100
  barre de menu, modèle Tea 425
  contenu (fondamental), insertion avec Tea 427
  décoration, ajout au modèle Tea 423
  mises à jour par le client 186
  mises à jour par le serveur 186
  modèles, opérations de remplacement de texte 429
  pages d'erreur pour les serveurs 141–143
  sortie, insertion du contenu d'un fichier dans 412
  traitement, poursuivre pendant la journalisation d'une exception (Tea) 410
  (voir aussi JSP)
pages d'erreur
  API Servlet 2.3, changements 555–558
  authentification à base de formulaire 228, 230
  directives JSP, utilisation 497–499
  exceptions, configuration 147
pages d'ouverture de session 234
  authentification à base de formulaire 228
    redirection du client vers 234
  autorisation personnalisée à base de formulaire 233
pages HTML
  compilation en classes Java 470
  création avec les servlets 12
  servlets HTTP, génération de 18–24
    données de formulaire, gestion 21
    gestion des requêtes HEAD 23
    page Bonjour tout le monde
      écriture 18–19
      exécution 19
    requêtes POST, gestion 23
paquetages
  conventions pour les noms 74
  javax.* 121
  javax.servlet et javax.servlet.http 6
<PARAM>, balises, applets 345
#param, directive WebMacro 441
  état de l'outil par défaut, définition 448
ParameterNotFoundException, classe 518
  code source 522
ParameterParser, classe
  code source 519–522
  envoi de données de forumlaire, utilisation dans 524
  renvoi du nombre de paramètres dans un entier 518
ParameterSnoop (exemple) 88–90
paramètres
  classe MultipartRequest, méthodes 113
  code 161
  codés en URL, dans le corps d'une requête POST 303
  dans la chaîne d'interrogation, accès 22
  données POST, codage 111
  effacement, instructions préparées 264
  éléments dans le caddie virtuel 195
  méthodes GET et POST 87
  requêtes d'inclusion, retrouver 359
  sortie, identification des types de données 283
  state 451, 509
paramètres d'initialisation
  clé, adresse IP et port du serveur 76
  clés, correspondance avec l'adresse IP et le port du serveur 76
  compteur lisant (exemple) 42
  définition pour les servlets dans le descripteur de déploiement 41
  établissement de connexions à une base de données avec 71
  servlets, obtention 71
  servlets, obtention des noms 72
paramètres d'initialisation du contexte 78–80
  connexions à une base de données 256
  informations de connexion à une base de données 255
  recherche du registre RMI avec 79
  registre RMI, recherche avec 79
  spécification dans le descripteur de déploiement 78
paramètres de requête 87–90
  accès par les beans embarqués dans les pages JSP 502
  ajout aux inclusions et renvois 505
  analyse 517–522
  japonais, lecture 554
  stockage dans une table de hachage 559
  Tea, obtention des informations sur 403
  utilisations 87
<param-name>, élément 638

<param-value>, élément 638
parenthèses, pour lever les ambiguïtés sur les variables WebMacro 436
parse() 446
#parse, directive WebMacro 440
partage d'informations, servlets 347–351
  interface ServletContext 348–351
    servlet SpecialSetter (exemple) 348
partage du contrôle, servlets 351–359
passerelle WAP 156
passivation (session) 553
pénalité de la première personne 494
performances
  classe mère WMServlet et 433
  contenu compressé et 183
  distribution de charge et 362
  JDBC, ouverture de connexions à une base de données 265
  mise en cache de plusieurs images et 182
  modèles WebMacro, amélioration avec la mise en cache de la sortie 452
  optimisation 544–546
  optimisation avec JIT et HotSpot 408
  persistance d'une servlet comme une instance d'objet, avantages 34
  rappels RMI, amélioration pour HTTP 292
  servlets, ajustement 544–546
  traitement des transactions, optimisation 269
Perl
  indépendance de la plate-forme 4
  langage de modèle de WebMacro, utilisation de 436
  PerlEx, amélioration des performances des scripts CGI 3
  programmes CGI, écriture en 2
persistance
  données non sérialisables sur plusieurs JVM 553
  servlets
    compteur (exemple) 35
    dans une JVM 33
    entre des requêtes sous forme d'instances d'objets 34
    entre les chargements (InitDestroyCounter) 44–47
  sessions, support du serveur pour 365
  variables locales et 38
personnalisation de l'affichage d'un ensemble de résultats 459
  classe BugIdCustomizer (exemple) 462
  classe DateCustomizer (exemple) 461
  classe NullCustomizer (exemple) 460
  classe NumberCustomizer (exemple) 461
personnalisation du comportement requête/réponse, API Servlet 2.3 550
personnalisée, authentification 230–237
personnalisées, bibliothèques de balises JSP
  installation et utilisation 512
personnalisés, filtres, WebMacro 453
phrase explicative (description du code d'état) 123
pilote API native et un peu de Java 252
pilote JDBC-ODBC 252
pilote réseau tout en Java 253
pilotes JDBC 251
  catégories 252
  classe, impossibilité de charge 258
  connexions à une base de données, obtention 253–255
  identifiants d'URL, inclusion dans 253
  liste, site web 253
  Sybase 265
plug-ins (conteneurs de servlets)
  débogage des servlets 538
PNG, format d'image 169
pointeurs, inexistance en Java 11
points-virgule, scriptlets et expressions JSP 499
portabilité des servlets 6, 10
  applets contre 10
portée d'application, JavaBeans 500
portée, JavaBeans
  définition dans une action <jsp:useBean> 501
  gestion par le serveur 500
  modifications dans l'implémentation de la classe, erreurs provenant de 504
ports
  connexions par socket, obtention pour 316
  registre RMI, écoute sur 325
  serveur, obtention 73
  serveur, obtention (servlet contre CGI) 69
  serveur, verrouillage d'une servlet à 76
  servlet écoutant sur, définition 334
  socket écoutant sur (thread Daemon) 313
POST, méthode 15
  appel de broadcastMessage() contre 333
  applets utilisant 289
  clients de discussion, poster des messages 328
  envoi d'un contenu à une servlet HTTP 110
  flots d'entrée, gestion avec 111
  formulaire avec URL d'action contenant une chaîne d'interrogation 89
  gestion par des servlets HTTP 17
  HttpMessage, envoi 303
  mise en cache (exemple Guestbook) 282
  mise en cache interdite 59
  objets sérialisés, envoi aux servlets 306
  paramètres 87
  restriction de l'accès aux servlets avec la balise <auth-constraint> 224

servlet Hello (exemple), gestion 23
type de contenu multipart/form-data 112
pourcentage, formatage pour un locale français 412
pratiques de programmation sûres, support des servlets pour 11
précompilation
JSP 494
modèles Tea 408
préférences
charset 385
charset et langue, servlet les utilisant pour accueillir le client 386–388
langue 384
préfixe de l'espace de nommage, correspondance avec la directive taglib 513
prepareCall() (Connection) 283
PreparedStatement, classe 264
préprocesseur du langage C, utilisation WebMacro de 436
PrimeSearcher, classe (exemple) 50–52
retour quand le dernier nombre premier est trouvé 55
primitifs, type de données Java, méthodes pour l'écriture 125
principals
interface Principal 225
nom, obtention à partir du certificat client 243
serveur contrôlant les rôles des 228
print() 125, 546
println() 125, 546
printMessages() 281
PrintStream, classe 19, 125
messages e-mail, écriture sur 523
trace de pile d'une exception, affichage 144
PrintWriter, classe 19, 110, 125
erreurs d'écriture de la sortie sur un client inexistant 150
FastWriter, utilisation à la place de 431
performances d'une servlet et 546
trace de pile d'une exception, affichage 144
problèmes de concurrence, éviter dans la migration de session 364
problèmes de gestion de la mémoire, évités dans Java 11
problèmes système, indisponibilité du serveur et 147
procédural contre déclaratif, langage de programmation 250
procédures stockées
base de données 282
précompilées et plus rapides 283
processus persistant pour chaque programme FastCGI 3
programmation orientée objet, CORBA et 292
programmes (externes), exécution à partir des servlets 529–533
programme finger 530–533
prolifération de processus, programmes CGI 3
<property>, balise, Apache Struts 512
propriétés
beans, contrôle dans les JSP 502
examen contextuel, outil de création de modèle Kettle 428
modèles WebMacro, gestion de 433
objet, recherche par introspection 436
variables (WebMacro), affectation de valeurs à 439
propriétés imbriquées (TeaServlet) 408
ProtectedResource, servlet (exemple), redirection du client vers la page d'ouverture de session 235
protocoles
HTTP (voir HTTP ; servlets HTTP)
HTTPS 303
IP 76
personnalisés (pour la communication applet/serveur) 290
requêtes, retour pour 106
réseaux sans fil, transport 155
SSL (Secure Sockets Layer) 239–246
TCP/IP 220
TLS (Transport Layer Security) 239
protocoles de transport de messages 523
pull du client 186
push du serveur 186–189
classe MultipartResponse 187–189
push piloté par servlet, utilisation dans WebMacro 430
PUT, méthode 16, 17

R

racine_document, répertoire 21
racine_serveur/conf/server.xml, fichier de configuration côté serveur 26
racine_serveur/webapps/ROOT, répertoire 25
modification des correspondances 26
racine_serveur/webapps/ROOT/WEB-INF/classes, répertoire 20
ramasse-miettes (Java) 11
rappels
clients de discussion RMI
notification de nouveaux messages 333
obtention de nouveaux messages avec 329
objets serveur, appel des méthodes d'applet 292
rapports des erreurs aux clients 144
codes d'état, utilisation 140

Reader, classe 110
readLine() 111
readObject() 306
<realm-name>, élément 224, 639
rebind() 323
rechargement de fichiers modèle, TeaServlet 408
rechargement des servlets 39
redirection de la sortie, commande finger 533
redirection des utilisateurs vers l'URL contenant l'identifiant de session 212
redirection du client vers la page d'ouverture de session 234
réécriture d'URL 212–213
  détection de l'identifiant de session 213
  fenêtres de navigateur multiples et 213
  filtre WebMacro pour 453
  identifiant de session unique, ajout 196
  identifiant de session, stockage avec 211
  information de chemin supplémentaire dans 196
  ShoppingCartViewerRewrite (exemple) 196–198
  ShoppingCartViewerSession (exemple) 217
  suivi de session avec 196–198
références
  à des fabriques de ressources externes 369
  à des images, retrouver 174
  aux composants EJB 368
registerOutParameter() 283
Registry, classe 323
réglage 544–546
Remote interface 316, 322
<remote>, élément 639
RemoteDaemonHttpServlet (exemple) 326
RemoteHttpServlet, classe 320–324
  auto-exportation 322
  DaytimeServlet, extension 317
  enregistrement/désenregistrement avec le registre RMI 322
  liaison au registre (code) 323
  RemoteDaemonHttpServlet contre 326
  UnicastRemoteObject contre HttpServlet, extension 322
removeAttribute() 204, 348
réponses 14, 123–153
  classes pour les accès à la servlet 18
  codes d'état 129–132
  connexions persistantes, utilisation 126
  contenu multimédia, envoi 155–189
  corps 124
  données caractères, écriture 125
  en-têtes 132–139
  erreurs et exceptions 139–151
  interface HttpServletResponse 19
  mise en tampon 127–129
    API Servlet 2.3, changements dans 559
  normales, envoi 124
  ordre des parties 124
  structure 123, 124
réponses validées 124, 128
  appel de getSession() avant 204
  appel de setStatus() avant 131
  erreurs, requêtes de clients arrêtées 150
  requêtes de renvoi et 353, 354
représentation arborescente, DOM 469
  remplissage avec un contenu HTML 476
représentation des données à l'utilisateur final 151
reprise sur erreurs d'une servlet 140
RequestDispatcher, interface 351, 567
  inclusions 355, 505
  méthodes 352
  renvois 506
requêtes 14, 87–90
  chaînes d'interrogation, attachement aux URL 15
  classes pour les accès à la servlet 18
  codage de caractères, information du serveur 553
  déterminer ce qui a été demandé 104
  distribution
    inclusion 355–359
    renvois 352–355
  en-têtes 106–110
    liste des 107
  examen pour le débogage de servlets 538–540
  flots d'entrée 110–121
  gestion de la terminaison 24
  HEAD, gestion 23
  information de chemin suppélmentaire 92–95
  interface HttpServletRequest 19
  méthodes
    GET et POST 15
    trouver 106
    (voir aussi GET, méthode ; POST, méthode)
  non-session et session, distribution de 364
  paramètres 87–90
    analyse 517–522
    utilisations 87
  personnalisées, création pour le débogage de servlets 540–543
  POST, gestion 23
  schémas 106
  Tea, obtention d'informations sur 403–406
requêtes d'inclusion
  action <jsp:include> 505

directives include, JSP 504
distribution 355–359
requêtes de redirection, renvois contre 355
requêtes de renvoi
action <jsp:forward> 506
distribution 352–355
servlets de renvoi, règles 353
requêtes non sûres (http) 106
requêtes SQL
ensembles de résultats 259–261
exécution 256–258
<res-auth>, élément 370, 639
réseau de confiance 239
réseaux (sans fil), couches de transport 155
réserve de connexions de bases de données 270–274
avantages pour les servlets 247
JDBC 2.0, intégrée 286
servlet Guestbook (exemple) 281
servlet OrderHandlerPool (exemple) 273
utilisation de init() 270
reset() 129
resetBuffer() 560
ResourceBundle, classe 385, 388
<resource-env-ref>, élément 548
<resource-ref>, élément 369, 640
<res-ref-name> element 640
<res-ref-sharing-scope>, élément 548
ressources
accès partagé ou exclusif à une référence de resource 548
information de chemin pour tous les contextes 559
lecture abstraite 98–100
protégées (voir authentification)
service 100–104
non sûr 100
pour le téléchargement 104
sûr 102
ressources abstraites, lecture
à partir d'un autre contexte 100
référence à la page d'accueil du serveur web 100
ressources dynamiques
cibles <location> 148
pages du code d'état de l'erreur 142
restriction des accès client 83
restriction des accès des utilisateurs aux pages web 84–85
restrictions d'exportation du gouvernement U.S., clés de chiffrement 238
<res-type>, élément 370, 640
ResultSet, interface 256, 259
ResultSetBuilder, classe 420
ResultSetMetaData, interface 259
colonnes, retrouver les noms et le nombre 260
génération de tables HTML à partir de ResultSet 259
ResultSetServlet, classe (exemple) 464–466
frontal HTML pour 467
ResultSetTable, classe (exemple) 463–464
Retry-After, en-tête 147
returnFile() 97, 150
returnURL() 99
RMI (Remote Method Invocation)
clients de discussion
notification de nouveaux messages 333
poster/recevoir des messages 329
communication applet/serveur 316–327
avantages et inconvénients 291
combinaison avec HTTP et non-HTTP 293
communication applet/servlet
DaytimeApplet (exemple), utilisation 324–327
DaytimeServlet (exemple), support 317–324
registre
enregistrement/désenregistrement avec 322
JNDI contre 366
où l'exécuter, décision 323
recherche avec les paramètres d'initialisation du contexte 79
servlets agissant comme clients 534
utilisation avec des applets et des servlets Java 11
rmic, compilateur RMI 318
RMIChatApplet (exemple) 338–343
rmiregistry, programme 323
<role-link>, élément 641
<role-name>, élément 641
caractère générique autorisant tous les rôles 560
rôles
autorisation basée sur 221
développement d'applications d'entreprise, J2EE 365
sécurité 221
API Servlet 2.3 560
rollback() 268
RSA, algorithme de chiffrement 238
<run-as>, élément 548
Runtime, classe 530
RuntimeException, classe 145
russe (cyrillique), alphabet, charset pour 377

S

saveState() 47
SAX, analyseur syntaxique 420
  conflits de version, éviter 554
schémas, requêtes de servlet 106
scriptlets 490
  JSP, problèmes avec leur utilisation 499
  page JSP, utilisation pour fixer le code d'état 497
  page toolview.jsp, utilisation dans 510
  remplacement par des balises personnalisées 512
Secure Sockets Layer, protocole (voir SSL)
sécurité 219–246
  API d'extension du serveur, risques 4
  attributs, nouveaux (API Servlet 2.3) 558
  authentification 220–237
    à base de formulaire 227–230
    HTTP 220–226
    personnalisée 230–237
  certificats numériques 220, 237–239
  cryptographie à clé publique 237–239
  fichier de propriétés de TeaServlet 408
  identifiants de session
    protection 211
    risques de leur utilisation 195
  identité de l'appelant d'un EJB, spécification 548
  préfixe /servlet/, (dés)activation 20
  programmes externes, lancement à partir des servlets 530
  requêtes GET et POST, éviter les risques de 16
  rôles 221
  servir des ressources abstraites 100–104
  servlets non sûres, gestion 140
  SSL 239–246
    authentification d'un client 240
    configuration 241
    configuration de l'authentification 242
    informations d'authentication, retrouver 243–246
<security-constraint>, élément 642
  balise <user-data-constraint> 241
  ordre dans le fichier web.xml 223
  protection d'un <web-resource-collection> 224
SecurityException, classe 287
<security-role>, élément 224, 642
  API Servlet 2.3 560
  ordre dans le fichier web.xml 223
<security-role-ref>, élément 225, 642
SELECT, instruction SQL 250
  liaison de plusieurs tables 250
Semiotek, Inc., moteur de modèle WebMacro 152
sendAndClose() 524
sendError() 132, 140
  images, renvoi 175
sendGetMessage() 302
sendPostMessage() 303
sendPostMessage(Serializable) 306
sendRedirect() 212
  forward() contre 352
séquences d'échappement (Unicode), caractères ISO-8859-1 374
sérialisation d'objet
  communication applet/servlet, utilisation dans 11
  communications applet/serveur, utilisation dans 291
  connexions HTTP, utilisation avec 293
Serializable, interface 203, 305
  applications distribuées, implémentation des objets de session liés 364
server-side includes (SSI), remplacement de 13
server-side JavaScript (SSJS) 5
ServerSnoop (exemple) 74
ServerSocket, classe 313
serveur de discussion 327–346
  implémenté comme une servlet, avantages 327
serveur X, création du Frame 168
serveurs
  algorithmes de chiffrement, choix de l'intégrité et de la confidentialité 242
  applets communiquant avec 287–346
    combinaison HTTP, non-HTTP et RMI 293
    connexions HTTP et par socket 288–290
    connexions non-HTTP 308
    CORBA-IIOP, utilisation 292
    HTTP à base d'objets 304–307
    HTTP à base de texte 297–304
    JDBC, utilisation 291
    RMI, utilisation 291, 316–327
    servlets set sérialisation d'objet 291
    sockets, non-HTTP 307–316
  authentification avec SSL 240, 243
  base de données des utilisateurs 222
  champs de formulaire cachés, support 195
  chemins de classes
    copie des fichiers .class pour les objets partagés 350
    modification pour inclure racine_serveur/webapps/ROOT/classes 319
  classes, séparation des classe de l'application web 554

codage des caractères des requêtes, information 553
composants JavaBean, travail avec 500
compression dynamique du contenu 183
configuration de la correspondance de contexte 26
conteneurs d'EJB 362
contenu WAP 159–160
déchargement de fichiers, rôle dans le 112
dégâts des servlets au, limitation 139
démarrage 20
détermination du port, servlets contre CGI 69
dysfonctionnements, persistance des servlets et 47
en-tête Content-Length, définition pour les connexions persistantes 126
en-tête Last-Modified, téléchargement d'une page par le navigateur 54
file-extension-to-mime-type, correspondances, connaissance de 95
gestion des exceptions 145
  retrouver la cause racine 146
HTTP
  incapacité à suivre des requêtes 191
  restriction des accès par 84
identifiants de session, sécurité des 195
implémentation de DOM, versions 487
implémentation du suivi de session 211
information de chemin supplémentaire, pour les programmes CGI 93
interaction d'une servlet avec (contre CGI) 70
obtention d'informations sur 73–82
  fichiers temporaires, écriture 75
  liste de tous les attributs courants 74
  logiciel serveur et ses attributs 73
  nom et numéro de port 73
  paramètres d'initialisation du contexte 78–80
  ServerSnoop (exemple) 74
  verrouillage d'une servlet au serveur 76–78
pages d'erreur
  configuration 141–143
  générée par sendError() 132
réécriture d'URL 196, 198, 212
répertoire racine_document 21
requêtes de client arrêtés, ne pas gérer les erreurs/exceptions pour 151
requêtes/réponses génériques, classes pour 17
respect des règles sur les applications distribuées 363
  serveur de discussion, construction 327–346
    implémenté comme une servlet 327
  servlets, rechargement 39
  sessions invalidées, incapacité à détecter 206
  SocketWatch, examen de la requête du client 538–540
  support du suivi de session 203
  valeurs de timeout d'une session et 208
  valeurs occasionnelles, génération pour l'authentification digest 221
serveurs d'applications
  CORBA, Inprise Application Server 4 8
  RMI, support des clients HTTP et par socket 293
  servlets, support de 6
  support du conteneur autonome 8
serveurs proxy 82
serveurs web
  Apache, utilisation avec mod_perl 3
  API d'extension du serveur 4
    servlets Java comme 5
  ASP, support de 5
  authentification de base 220
  authentification du client, gestion 230
  classes de servlet, utilisation 7
  code 404 Not Found 124
  connexion, création avec HttpClient 541–543
  conteneurs de servlets et 7
  exécution avec les servlets dans une JVM 34
  iPlanet/Netscape 7
  partage d'informations entre 349
  passerelles WAP pour les périphériques WAP 156
  PerlEx, amélioration des performances des scripts CGI/Perl 3
  programmes FastCGI, incapacité de travaille avec 3
  référence à la page d'accueil, obtention 100
  scripts CGI, incapacité d'interagir avec 3
  sécurité 220
  support des conteneurs autonomes 7
  support du suivi de session, solution sans cookies 212
  (voir aussi serveurs d'applications ; serveurs)
serveurs web multiprocessus, exécution d'une JVM externe 34
serveurs web multithread à processus unique dans d'autres langages, embarquement d'une JVM 34
service de ressources 100–104
  non sûr 100
  pour le téléchargement 104
  sûr 102

service()
classe HttpServlet 17, 23, 106
classe Servlet, gestion des exceptions 145
réutilisation des objets de base de données 265
servlets implémentant SingleThreadModel 48
servlet Annuaire téléphonique (exemple) 265
servlet Bonjour tout le monde 124
ECS, génération avec 456
HelloJapan (exemple) 378
HelloJapanReader (exemple) 380
HelloSpain (exemple) 372
date localisée, affichage 375
échappements Unicode, utilisation 374
entités caractères numériques, utilisation 373
HTTP 18–20
écriture 18
exécution 19
langue préférée du client, utilisation (HelloBabel) 386–388
servlet Hello (exemple) 21–23
requêtes HEAD, gestion 23
requêtes POST, gestion 23
sortie multilingue (exemple HelloRosetta) 382–384
servlet Caddie virtuel
ajout d'éléments à 192
retrouver un élément 192
ShoppingCartViewerCookie (exemple) 201–202
ShoppingCartViewerHidden (exemple) 193–195
ShoppingCartViewerRewrite (exemple) 196–198
ShoppingCartViewerSession (exemple) 217
valeurs de timeout, choix 209
servlet Livre d'or (exemple) 277–282
utilisation de CacheHttpServlet 56–59
servlet Panneau d'affichage, lecture de longues chaînes ASCII 284
servlet Traitement des commandes
utilisant de la classe ConnectionPool 273
utilisation des transactions (exemple OrderHandler) 268
servlet transactionnelle avec réserve de connexions (OrderHandlerPool) 273
<servlet>, élément 643
balise <security-role-ref> 225
entrées de <init-param> dans 42
<load-on-startup>, ajout à 52
Servlet, interface 16, 568
<servlet-class>, élément 643
ServletConfig, interface 43, 71–73, 570
ServletContext, interface 73, 571–576
distributeur de requêtes, obtention 351
liste de tous les attributs courants, serveur et servlet 74
noms et liste de toutes les ressources dans 559
partage d'informations avec 348–351
réserve d'instances de HttpSession 203
retrouver les informations d'initialisation du contexte 78
version de l'API Servlet, retour 80
ServletContextAttributesListener, interface 551
ServletContextEvent, classe 551
ServletContextListener, interface 551
ServletDebugger 543
ServletException, classe 145, 146, 576
affichage des exceptions lancées au serveur 148
comme enveloppe de toute exception ou erreur 146
sous-classe UnavailableException 146
ServletExec, conteneur ix, 9
ServletInputStream, classe 110, 578
<servlet-mapping>, élément 644
<servlet-name>, élément 644
ServletOutputStream, classe 125, 182, 185, 546, 579
écriture de la sortie sur un client inexistant 150
ServletRequest, interface 17, 73, 85, 580–586
locales, renvoi du préféré 393
obtention du chemin de contexte 95
paramètres, renvoi 87
ServletResponse, interface 17, 586–589
contrôle du tampon de réponse 128
locale, spécification 393
taille du tampon, définition 127
servlets 5
accès 20
agissant comme clients RMI 534
ajout d'attributs aux requêtes 121
applets communiquant avec 287–346
classe HttpMessage 303
RemoteDaemonHttpServlet (exemple) 326
servlets dans le tiers intermédiaire, gestion 291
avantages 10–12
efficacité et endurance 11
élégance du code 12
extensibilité et souplesse 12
intégration au serveur 12
portabilité 10
pratiques de programmation sûres 11

puissances des API Java fondamentales 10
sûreté 11
clients arrêtant les requêtes sur 149
collaboration 347–359
de type WebMacro, invocation 435
débogage 536–544
définition 1
fichiers de classes, répertoire pour 20
filtres, liaison à 551
génération de graphiques 172
génériques contre HTTP 72, 124
indisponibilité temporaire 147
intégration avec J2EE 365–370
noms, obtention 72
paramètres d'initialisation 71
obtention des noms 72
(voir aussi paramètres, initialisation)
réponse, structure 124
sauver l'état 44
sécurité (voir authentification ; sécurité ; gestionnaires de sécurité)
support 6–10
conteneurs additionnels 9
conteneurs autonomes 7
conteneurs embarquables 9
verrouillage au serveur 76–78
WebMacro, création 430
servlets d'arrière-plan 492–494
contrôle avec des directives JSP 495–500
jspInit() et jspDestroy() 495
moment de la conversion 505
servlets d'entreprise 361–370
distribution de la charge 361–365
intégration avec J2EE 365–370
composants EJB, références à 368
distribution des servlets 370
entrées de l'environnement 366–368
fabriques de ressources externes, références à 369
servlets HTTP 13–31
API Servlet 12, 16–18
applications web, correspondances de servlet 28–31
compteur général (exemple) 38–39
compteur synchronisé simple 35
cycle de vie 33–67
chargement au démarrage 52
compteur lisant des paramètres d'initialisation (exemple) 42
init() et destroy() 40–47
mise en cache côté client 54–56
mise en cache côté serveur 56–67
modèle mono-thread 48–50
rechargement 39
SimpleCounter (exemple) 35
traitement en arrière-plan 50–52
génération de pages 18–24
données de formulaire, gestion 21
page Bonjour tout le monde
écriture 18–19
exécution 19
requêtes HEAD, gestion 23
requêtes POST, gestion 23
génériques contre 72
les bases, HTTP 13–16
méthodes 16
correspondance avec les variables d'environnement CGI 70
GET et POST 15, 17
PUT et DELETE 17
ordre des réponses 124
paramètres de requête, obtention 87
protocole et numéro de version 106
servlets génériques contre 124
(voir aussi GenericServlet, classe)
URI de requête contre URL de requête 105
servlets longues, arrêt du traitement avant la fin 150
servlets sûres, gestion par le gestionnaire de sécurité 140
ServletUtils, classe 97, 99, 103
obtention de la trace de pile sous forme d'un String 144
ressource, servir de façon non sûre 102
session, attribut JSP 497
session, identifiants 211–213
génération 198
réécriture d'URL pour l'inclusion 212
suivi de session avec 195
validité de, déterminaison 213
session, migration 364
auditeur pour 553
session, persistance 365
SessionBindings, servlet (exemple) 216
<session-config>, élément 644
sessionCreated() 552
sessionDestroyed() 552
sessionDidActivate() 553
session.invalidate() 230
sessions
attribut session, JSP 496
initialisation et destruction, auditeurs 551
outil du contexte de WebMacro 437
Tea, manque de facilité d'accès 406
sessions utilisateurs (voir suivi de session ; sessions)
SessionSnoop, servlet (exemple) 213
<session-timeout>, élément 207, 644
SessionTimer, servlet (exemple) 207

SessionTracker, servlet (exemple) 204
sessionWillPassivate() 553
#set, directive WebMacro 433, 439, 443
setAttribute() 74, 121
  auditeur de session, liaison/déliaison 217
  interface HttpSession 204
  interface ServletContext 348
  requêtes de renvoi, utilisation dans 352
setAuthorization() 303
setAutoCommit() 268
setBackground() 457
setBufferSize() 127
setCharacterEncoding() 518, 553, 559
setContentLength() 126
setContentType() 165
  interface HttpServletResponse 19
  interface ServletResponse 125
  setLocale(), appel avant 393
setFilterConfig() 549
setHeader() (MailMessage) 523
setLocale() (ServletResponse) 393
setMaxInactiveInterval() 207, 209
setName() 504
setNextMessage() 329
setSize() 502
setStatus() 131
  utilisation contre sendError() 140, 143
setString() (PreparedStatement) 264
setToolsFile() 511
SGBDR (Système de gestion de base de données relationnelle) 249
  (voir aussi connexions aux bases de données ; bases de données relationnelles)
  indices, ensembles de résultats 260
  pilotes JDBC 252
shells, redirection de la sortie 533
Shimari, projet 152
signatures numériques 238
  applets sûres 288
Silverstream Application Server 9
Simple Mail Transfer Protocol (SMTP) 523
SimpleChart, servlet (exemple) 169–171
SimpleCounter (exemple) 35
  extension pour la lecture des paramètres d'initialisation 42
SimpleResultSetTable (exemple) 457–459
simulateurs de périphériques, WAP 158
simulateurs de téléphones WAP 158
SingleThreadModel, interface 48–50, 589
size, propriété des beans 502
sleep() 188
<small-icon>, élément 645
SmtpClient, classe 523
SnoopManipulation, classe (exemple) 478
SocketChatApplet (exemple) 337
SocketException, classe 338
SocketWatch, classe (exemple) 538–540
Solaris, démarrage d'un serveur sur 20
sortie
  codée 378–381
  exception (exemple) 145
  mise en tampon pour améliorer les performances des servlets 545
  modèles WebMacro
    capture sous forme de chaîne des caractères 432
    mise en cache 452
  non-HTML, gestion avec les échappements Unicode 374
  redirection pour la commande finger 533
  ressources incluses, contraintes 359
  servlet Guestbook (exemple) 281
souplesse des servlets 12
<SPAN>, éléments
  englobant le texte à remplacer 470
  fichier de maquette HTML 482
SpecialSetter, servlet (exemple) 348
  autres servlets accédant et affichant la spécialité du jour 349
spéciaux, effets sur des images
  niveaux de gris 179–181
SQL (Structured Query Language)
  ANSI SQL-2 Entry Level 250
  caractères de contrôle, gestion correcte 264
  ensembles de résultats 259–261
    méthodes pour retourner les types de données 260
  exceptions, gestion 258
  exécution de requêtes 256–258
    servlet DBPhoneLookup (exemple) 257–258
  instructions
    génération de tables HTML 262
    mise à jour de la base de données 262
    procédures stockées 282
    regroupement en transactions 268
    SELECT 250
    stockage dans des variables staitiques finales 281
  instructions préparées 264
    réutilisation 266
  procédures stockées
    précompilées et plus rapides 283
  types de données SQL3, support dans JDBC 2.0 286
SQLException, classe 258
sql.properties, fichier 254
  classe ContextProperties, utilisation en place de 256
squelette de mot de passe 233

squelette, classes (voir classes stub et squelette, RMI
SSL (Secure Sockets Layer) 106, 239–246
authentification d'un client 240
configuration de l'authentification 242
certificats client, basée sur 242
configuration de la sécurité 241
identifiants de session 211
informations d'authentification, retrouver 243–246
(voir aussi HTTPS, protocole)
standard C++/Java, indices 260
start() 432, 451
startResponse() 188
state, paramètre 451, 509
Statement, classe 256
mise à jour de la base de données 262
stop() 313, 432
désapprouvée, pour l'arrêt des threads 52
StringBuffer, classe 544
Structured Query Language (voir SQL)
Struts (voir Apache Struts)
<struts:iterate>, balise 516
<struts:property>, balise 516
substitution, application, XMLC 470
suivi de session 191–218
anonyme
utilisation de la réécriture d'URL 196–198
utilisation des champs de formulaire cachés 193–195
applets avec le Java Plug-In, travail avec les limites 303
authentification de l'utilisateur, inconvénients 193
basé sur une applet 211
champs cachés, utilisation 193–195
cookies persistants, utilisation 198–203
dans l'optimisation du traitement des transactions 270
identifiants de session, utilisation pour 195
implémentation par le serveur web 211
perte d'informations à l'invalidation 206
servlets, classe CacheHttpServlet et 59
servlets, fonctionnalités intégrées 203–218
création ou obtention de la session courante 204
cycle de vie d'une session 206
définition du timeout 206–209
événements de liaison de session 215–217
interface HttpSession 203
liaison d'un objet de session 204
méthodes du cycle de vie 209
objets liés, renvoi des noms 204
réécriture d'URL, utilisation 212–213
SessionSnoop (exemple) 213
SessionTracker (exemple), utilisation du compteur de hits 204
ShoppingCartViewerSession (exemple) 217
suppression des objets des sessions 204
transactions, gestion avec 274–277
sujet, définition dans les messages e-mail 523
Sun Microsystems
bibliothèque JIMI (Java Image Management Interface) 169
division Java Software, noms d'attributs réservés à 74, 121
Java Web Server 8
JavaServer Pages 152
(voir aussi JSP)
paquetages com.sun.* 74
super.destroy() 44, 313
RemoteDaemonHttpServlet (exemple) 326
servlets distantes, terminaison de la communication RMI 317
super.init() 313
débogage de servlet, vérification dans 544
super.init() et super.destroy() 313
super.init(config) 43
sûres, applets 287
sûreté
des servlets 11
des threads 266
réutilisation d'instructions préparées 266
des types de données Java 11
des types de données, sérialisation d'objets 291
Sybase, pilote JDBC 265
synchronisation
blocs de code 36
dans les environnements distribués 363
doPost(), optimisation du traitement des transactions 270
état de l'application web avec la base de données 552
limitation pour les performances des servlets 545
sections de code utilisant les objets partagéss 266
SingleThreadModel et 48, 50
transactions de bases de données et 267
synchronized, mot-clé, ajout à doGet() 37
System, classe 498
débogage des problèmes de chemin de classes 543
systèmes à clés symétriques, cryptographie 238

systèmes d'exploitation
démarrage d'un serveur 20
portabilité des servlets sur 10
systèmes, lancement d'exceptions distantes pour les erreurs réseau 317

T

table de hachage, stockage des images converties dans 182
tableaux
d'objets TableCustomizer 463
d'octets 111
définition dans WebMacro 439
type de données ARRAY de SQL3, support dans JDBC 2.0 286
TableCustomizer, classe (exemple) 459
tables de bases de données 249
employés, affichage des noms et des numéros de téléphone 257–258
instruction SELECT, liaison de plusieurs 250
Tag Library Descriptor (TLD), fichier
installation 512
taglib, directive JSP 513, 516
<taglib>, élément 645
ajout au descripteur de déploiement 512
<taglib-location>, élément 512, 645
Taglibs (Apache) 511
<taglib-uri>, élément 512, 646
taille, tampon de réponse 127
TCP/IP, manque de sécurité 220
TD, classe 457
Tea 152, 399–428
administration 406–410
erreurs et exceptions, gestion 410
fichier de configuration (exemple) 407–408
messages d'erreur de compilation 409
vérification du bon chargement de l'application 416
applications 410–416
classe NameApp (exemple) 413
classe NameContext (exemple) 414
écriture 413–416
fonctions de traitement du texte 410
outils de création de contenu, liste 416–427
information sur la requête, obtention 403–406
langage 400
instructions 405
TeaServlet
administration 406–410
fichier de propriétés (exemple) 407–408
fichier de propriétés, ToolApp (exemple) 418
téléchargement et installation 399, 401
technologies JIT et HotSpot, optimiser les performances 408
téléchargement de ressources sur les navigateurs 104
telnet, programme, établissement d'une connexion au serveur avec 540
TemplateExpireTime, propriété 446
TemplatePath, paramètre
configuration 434
fichier cible de #parse, inclusion dans 440
test/débogage de servlets 536–544
à l'aide d'un débogueur standard 537
ajustement des performances 544–546
examen de la requête du client 538–540
journaux, contrôle 536
navigateurs, utilisation dans 544
outil tiers, utilisation 543
problèmes de chemin de classes 543
requête de client personnalisée, création 540–543
text/vnd.wap.wml, type de contenu 164
texte
contenu, périphériques WAP 157
conversion en octets 398
éditeurs internationalisés 380
fonctions de traitement, Tea 410
formatage avec le langage Tea 400
longues chaînes, gestion par la servlet Panneau d'affichage 284
modèles WebMacro, inclusion/suppression avec #use 441
régions, modèles Tea 401
remplacement dans les modèles de pages web 429
Thawte Consulting (autorité de certification) 239
ThreadDeath, exception 313
threads
classe DaytimeConnection (exemple) 310
client, manipulant des variables non locales d'une servlet 36
créés par des servlets, persistance 34
gestion de requêtes concurrentes multiples 11
gestionnaire, pour les clients 313
initialisation et destruction 313
instances d'objet, manipulation 35
interface SingleThreadModel 48–50
servlets, gestion avec 6
synchronisation 35–38
traitement en arrière-plan 50–52
threads d'arrière-plan, arrêt 52
threads démon
arrêt 313
création dans l'init() d'une servlet 313

démon, définition 308
Throwable, classe 146
Tidy, analyseur syntaxique, conversion de HTML en DOM 470
tiers intermédiaire
EJB, utilisation comme 249
servlets dans 248
applets, connexion à 291
timeout personnalisé, définition pour l'utilisateur 209
timeout, définition pour les sessions 206–209
choix de la bonne valeur 208
configuration individuelle 207
par défaut, applications web 207
SessionTimer (exemple) 207
TLS (Transport Layer Security), protocole 239
Tomcat Server
base de données des utilisateurs 222
classe HttpMessage, emplacement 303
débogage des servlets avec 537
identifiant de session 211
implémentation de DOM 487
moteur JSP, Jasper 489
servlet d'arrière-plan, hello1.jsp 492–494
Tool, classe (exemple) 421–423, 487, 510
WebMacro, ToolServlet (exemple) 450
ToolApp, classe (exemple) 416–420
fichier de données exemple 418
ToolBean, classe 510
ToolContext, classe (exemple) 420
Toolkit, classe 168, 172
ToolServlet, classe (exemple) 450–452, 513–515
toolview1, modèle (exemple) 423
toolview2, modèle (exemple) 423
ToolViewServlet (exemple), classe de manipulation XMLC 485–488
toolview-tag.jsp, page (exemple) 515
Town (API de base de données) 286
TRACE, méthode 16, 17
traces de la pile
exceptions et 144
exceptions, API Servlet 2.3 556
journalisation des erreurs 144
pages d'exception et 149
traitement en arrière-plan 50–52
arrêt 52
(voir aussi servlets d'arrière-plan)
transactions 267–277
distribuées, JDBC 2.0 286
JDBC 268
validation/annulation 268
JTA dans J2EE 365
réserve de connexions 270–274
servlet OrderHandlerPool (exemple) 273–274
suivi de session, gestion avec 274–277
traitement optimisé 269
validées 267
transformation (XML en HTML), avec les servlets 12
Transport Layer Security (TLS), protocole (site web) 239
<transport-guarantee>, élément 242, 646
try/catch/finally, construction 150
type, attribute (beans) 501
types de contenu
application/octet-stream 104
application/x-www-form-urlencoded 111
définition pour la réponse d'une servlet 125
fichiers d'image 165
fichiers déchargés, obtention pour 114
fichiers WML 159
image/gif 168
multipart/form-data 112
objets Java sérialisés et 305
pages JSP, spécification 495
réponses, définition pour la sortie de WebMacro 433
requêtes POST, application/x-www-form-urlencoded 303
vérification pour les flots d'entrée des requêtes 111
types de données
chaînes de caractères
paramètres d'initialisation comme 44
(voir aussi chaînes de caractères)
chaînes de texte longues et binaires, gestion de la base de données 284–285
contrôle fort, servlets 69
entrées de l'environnement 367
instances d'objet de servlet, gestion dans un thread 36
long, stockage des heures 55
paramètres de requête, conversion en Tea 403
paramètres de sortie, identification 283
primitifs, méthodes pour l'écriture 125
sérialisation d'objet, sûreté 291
SQL, méthodes pour retourner à partir d'une base de données 260
SQL3, support dans JDBC 2.0 286
sûreté du langage Java 11
valeurs d'en-tête 108
variables WebMacro 439
types MIME 95
dans l'en-tête Accept (HTTP) 14
image/gif 168
objets sérialisés non représentés 305

U

UnavailableException, classe 146, 148, 589–591
unbind() 324
UnicastRemoteObject, classe 322
  exportation d'objets distants 322
Unicode 374
  caractères 16 bits, sortie sur le client 381
  charset natif, conversion en 396–398
  conversion dans un codage spécifique au locale 19
  conversion de contenu statique, éviter dans WebMacro 431
  échappement des caractères 374, 396, 659–664
  jeu de caractères UCS-2 381
  jeu de caractères UTF-8 381–384
Uniform Resource Identifiers (voir URI)
Uniform Resource Locators (voir URL)
Unify eWave, conteneur ServletExec 9
Universal Character Set, forme à 2 octets (UCS-2) 381
Unix, systèmes
  pair natif du Frame, création 168
  programme telnet 540
  shells, redirection de la sortie 533
UnsafeViewResource (exemple) 100
UnsupportedEncodingException, classe 110, 125, 379
update() 333
UPDATE, instruction SQL 262
UploadTest (exemple) 114
URI
  chemin de préfixe, correspondance de l'application web avec 25
  chemins de contexte de servlet 350
  chemins vers des composants 351
  élément <taglib-uri> 513
  nom de la servlet invoquée 105
  paramètre de chemin, liaison avec la ressource réelle 98
  préfixe du contenu, obtention 95
  requête, obtention pour 105
  ressource problématique, traçage pour les exceptions 556
  URL et 105
URL
  absolues et relatives, distribution de requêtes 351
  ajout de chaînes d'interrogation 15, 21
  configuration pour l'accès aux servlets 20
  HttpMessage, communication avec 302
  inclusion de contenu avec une directive WebMacro 440
  information de chemin supplémentaire, inclusion 92
  invocation de servlets enregistrées 28
  JDBC 253
  noms de servlet dans 105
  originale, reconstruction à partir des objets de requête 559
  pages d'ouverture de session et d'erreur, authentification à base de formulaire 228
  protocole HTTPS 240
  relatives, gestion dans la distribution de requêtes 355
  requête, obtention 104
  serveur incapable de localiser 124
  URI et 105
  ViewFile (exemple), gestion des 98
  réécriture (voir réécriture d'URL)
URL, classe
  applets utilisant pour la communication avec le serveur 289
  cache des connexions, incapacité de contrôler 303
  connexion par socket 302
URLConnection, classe
  applets utilisant pour la communication avec le serveur 289
  connexion par socket 302
URN (Uniform Resource Name) 105
US-ASCII, caractères
  accélération de la sortie dans WebMacro 431
#use, directive WebMacro 441
User-Agent, en-tête 183
<user-data-constraint>, élément 241, 647
UTF-8 (UCS Transformation Format, forme à 8 bits) 381–384, 396
utilisateur, authentification (voir authentification)
utilisateurs, obtention des informations sur 84–85

V

valeur alpha (transparence), pixels 181
valeurs
  cookies, initiales 198
  cookies, syntaxe 200
  objets de session, modification des noms de méthodes 204
  timeout, servlet Caddie virtuel 209
valeurs de couleur RVB, conversion en niveaux de gris 181
valeurs de pixel (table des couleurs), conversion en niveaux de gris 181
valeurs null
  champs de base de données 261
  cookies, retour pour 199

ensembles de résultats SQL 260
filtrage de l'affichage, WebMacro 453
lots de ressources, indication de non-correspondance 388
NullCustomizer (exemple) 460
paramètre d'état 511
type d'autorisation, gestion 559
valeurs occasionnelles 221
validation (utilisateurs), authentification de base et à base de formulaire 230
validation de fichiers XML 28
ordre des balises, fichier web.xml 29
validité, déterminaison pour les identifiants de session 213
valueBound() 215, 217
valueUnbound() 215, 217
variables
de classe (statiques) 38
JSP, déclaration statique ou d'instance 494
liste des outils, types de 448
locales et non locales, manipulation par le client 36
locales, gestion Tea des 405
scriptlet, prédéfinies 490
WebMacro 433
directive #set, attribution de valeurs avec 439
filtrage 452
syntaxe 436
variables d'instance
changement en locales 38
déclaration dans les JSP 494
environnements distribués, inadaptation pour 362
persistance de 35
variables statiques 38
déclaration dans les JSP 494
déclaration erronée dans une scriptlet 499
environnements distribués, inadaptation pour 362
stockage des instructions SQL 281
Vary, en-tête 185
servlet HelloBabel (exemple) 387
vecteurs
contenant des objets sérialisables 305
vecteur socketClients, ajout de nouveaux clients de discussion à 333
Vector, classe, synchronisation interne 545
Velocity, environnement de développement (clone de WebMacro) 429
vente de burritos, utilisation du contexte de servlet 348
VeriSign Corporation 239
verrouillage d'une servlet à un serveur 76–78
verrous (thread), sur une instance d'objet 37
VersionDetector, classe 80–82
versions
cookies, définition 199
protocole de requête, retour pour 106
VersionSnoop (exemple) 82
vidage des tampons JSP 496
ViewFile (exemple) 96
amélioration avec les codes d'état 132
bloc finally, garantie de la fermeture du FileInputStream 150
journalisation des erreurs 143
trace de la pile d'une IOException, fournir 144
ViewResource, servlet (exemple) 165
servir des ressources abstraites de façon sûre 102
ViewResourceCompress, servlet (exemple) 184–186
Village (API de base de données) 286
Visual Engineering, KavaChart, paquetage 172
vue (architecture MVC) 151

W

wait() 333
Walt Disney Internet Group (WDIG) 152, 399
(voir aussi Tea)
WAP (Wireless Application Protocol)
contenu dynamique 160–165
contenu, servir 159–160
cookies, acceptation par les périphériques 203
kits de développement et simulateurs (sites web) 158
piles, Web contre 156
simulateurs de périphériques 158
WAP Forum 155
WML et 155–164
WAR (Web Application Archive), fichiers 25
dépendances avec les fichiers JAR, expression dans 554
getPathTranslated(), ne fonctionnant par avec 98
wasNull() 261
WBMP (Wireless Bitmap), format d'images 157
WDP, WTLS, WTP et WSP, protocoles de transport 155
web
passerelle WAP 156
tiers intermédiaire, implémenté avec des servlets 248
web application archive (voir WAR, fichiers)
Web Application Environment (WAE) 155
web, navigateurs (voir navigateurs)
web, pages (voir JSP ; pages)

WebContext, classe 431
liste des outils, placement sous 452
WebLogic Application Server 8
WebMacro 152, 429–453
application outil 447–452
directives 439–441
environnement 430–433
modèle Hello 430
filtres 452
personnalisés 453
installation 433–438
fichier de propriétés 433–435
outils du contexte 436–438
langage de modèle 436
licence 429
modèles 442–447
traitement 446
servlet MacroView (exemple) 444–446
WebMacro, interface 431
WebMacroException, classe 432
<web-resource-collection>, élément 224, 648
<web-resource-name>, élément 648
WebSite Professional (web server) 8
WebSphere Application Server 8
web.xml, fichiers
ordre des balises requis 223
(voir aussi descripteur de déploiement ; DTD)
<welcome-file>, élément 649
<welcome-file-list>, élément 649
utilisation avec du contenu non-WAP 160
WHERE, clause, utilisation avec l'instruction DELETE 251
Windows, systèmes
démarrage d'un serveur sur 20
Kettle, environnement de création de modèles 428
programme telnet 540
shells, redirection de la sortie 533
Wireless Bitmap (WBMP)
format d'images monochromes pour les graphiques 157
types de contenu 159
WM, classe 431
WML (Wireless Markup Language) 151, 155–164
application carte des consommations 157–158
document agissant comme une mini carte des boissons 157
formulaires 161
navigateurs web, support pour 158
ressources, sites web 164
WMLScript, langage 157
ressources, sites web 164
types de contenu 159
WMServlet, classe 451
WMServletHello, classe (exemple) 432–433
World Wide Web Consortium (W3C)
Jigsaw Server 8
processus de normalisation de XML 27
write() 125, 432
WWW-Authenticate, en-tête 107
description des certificats souhaités 231

X

X.509v3, certificat client, servlet retrouvant 243–246
Xerces 476
XML 27
analyseurs syntaxiques supportant différentes versions de DOM ou SAX 554
API JDOM, manipulation avec 420
application WML (drinks.xml) 157–158
bien formé, action JSP et 502
Cocoon (environnement de publication web) 153
compilation de documents en classes Java 469
contenu de modèle, utilisation dans 430
descripteurs de déploiement 26
(voir aussi descripteurs de déploiement)
DTD (Document Type Definition) (voir DTD)
éditeurs 28
espaces, traitement 560
fichier de propriétés de TeaServlet et 408
outils de création de contenu basés sur 153
préfixe de l'espace de nommage, correspondance avec la directive taglib 513
site web d'informations du W3C 27
ToolApp (exemple), fichier de données exemple 418
transformation en HTML 12
XML Compiler (voir XMLC)
XML DOM (Document Object Model), représentation arborescente 469
XMLC 153, 469–488
application de substitution 470–476
Hello.java, code source autogénéré 472–476
simple 470
application outil 480–488
classe manipulation 485–488
fichier de contenu HTML 482–484
fichier de modèle HTML 480–482
caractères spéciaux dans un élément, échappement 488
classes de manipulation 476–480

maquette de liste de langues 478–480
page Hello 476
fichier JAR, installation pour les classes 477
installation 470
modèle d'objets DOM, problèmes avec 480
téléchargement (site web) 469
xmlc, compilateur 470
XSLT, utilisation dans la gestion du contenu 153

Y

Yahoo, suivi de session dans Mon Yahoo! 206

Z

Zeus Web Server 8
ZIP et GZIP, formats de compression 183
ZipOutputStream, classe 185

À propos des auteurs

Jason Hunter est technologue senior chez CollabNet (*http://collab.net*), une société fournissant des outils et des services pour la collaboration à des projets en *open source*. En plus d'être l'auteur de *Servlets Java, Guide du programmeur*, il est l'éditeur de Servlets.com, le créateur de la bibliothèque com.oreilly.servlets, un contributeur du projet Apache Jakarta qui a créé Tomcat (il était encore chez Sun quand il a rejoint ce projet), un membre du groupe d'experts responsable du développement des API Servlet/JSP et JAXP, et il a un siège, en tant que représentant de l'Apache Software Foundation, au JCP Executive Committee qui supervise la plate-forme Java. Il écrit aussi des articles pour JavaWorld et tient des conférences à plusieur salons sur la programmation et les logiciels libres. Plus récemment, il a été le co-créateur de la bibliothèque à sources ouvertes JDOM (*http://jdom.org*) qui optimise l'intégration de Java et de XML et il conduit le groupe d'experts responsable du développement de JDOM.

Jason était précédemment le directeur technique de K&A Software, une société spécialisée dans les formations et le consulting Java. Ses services en tant que gourou Java ont été offerts à des dizaines de sociétés, y compris Sun Microsystems. Avant cela, il a travaillé chez Silicon Graphics où il était responsable du développement (et de la destruction) de toutes sortes de technologies web.

Jason a obtenu un diplôme en informatique *cumma cum laude* de l'Université Willamette (Salem, Oregon) en 1995. Il a commencé à programmer en Java à l'été 1995 et il s'est impliqué dans les servlets et les technologies liées aux extensions du serveur depuis décembre 1996. Si, par miracle, vous ne le trouvez pas à son travail, c'est qu'il est probablement en train de se balader dans les montagnes.

William « Will » Crawford est impliqué dans le développement web depuis 1995. Il a travaillé au Children's Hospital Informatics Program à Boston, où il a participé au développement du premier système d'enregistrement médical électronique basé sur le web, et il a été impliqué dans certaines des premières utilisations de Java au niveau de l'entreprise. Il a été consultant pour des projets de développement d'intranets pour, en autres, le Children's Hospital, le Massachusetts General Hospital, le Grigham and Women's Hospital, la Boston Anestesia Education Foundation et le Harvard Medical Center.

Will dirige actuellement l'équipe de développement produit de Invantage, Inc., une startup de Cambridge, dans le Massachusetts, qui développe des outils intranet basés sur Java pour l'industrie pharmaceutique. Pendant son temps libre, il est photographe amateur, écrivain, et suit un Bachelor's of Economics à l'Université de Yale.

Colophon

Notre look est le résultat de commentaires des lecteurs, de notre propre expérience, et de la réaction de nos revendeurs. Nos couvertures différentes sont complémentaires de notre approche différente des sujets techniques, donnant personnalité et vie à des sujets potentiellement arides.

L'image sur la couverture de *Servlets Java, Guide du programmeur, 2e édition*, est un ours d'Europe.

On recense huit espèces d'ours : le panda, l'ours brun, l'ours polaire, l'ours à collier, l'ours malais ou ours des cocotiers, l'ours lippu, l'ours à lunettes.

La majorité des ours vit dans l'hémisphère nord. Leur anatomie est fort semblable, mais la taille varie considérablement : de 150 livres à 1 800 selon l'espèce. De même l'habitat et le régime alimentaire sont proches d'une espèce à l'autre. Mis à part l'ours blanc, les ours sont omnivores. L'ours peut vivre entre 25 et 40 ans, ceux en captivité ont la durée de vie la plus longue.

Colleen Gorman a été la rédactrice en chef technique et Norma Emory a été la correctrice des épreuves de l'édition originale. Catherine Morris et Leanne Soylemez ont procédé au contrôle de qualité. Frameworks Consulting a fourni l'assistance à la production. Ellen Troutman-Zaig a écrit l'index.

La couverture a été conçue par Hanna Dyer, d'après une série de Edie Freedman. L'image est une photographie de Kevin Thomas modifiée avec Adobe Photoshop par Michael Snow. La mise en page de la couverture a été réalisée par Emma Colby avec QuarkXPress 4.1 en utilisant la police Bodoni Black de URW Software et Bodoni BT Bold Italic de Bitstream.

David Futato a conçu la mise en page intérieure de la version originale de ce livre, d'après un modèle crée par Nancy Priest. Judy Hoer a converti en FrameMaker 5.5.6 les fichiers provenant de Microsoft Word en utilisant les outils créés par Mike Sierra. La police utilisée pour les en-têtes est Bodoni BT ; la police pour le texte est New Baskerville et celle du code est Constant Willison. Les illustrations ont été créées avec Macromedia Freehand 8 et Adobe Photoshop 5 par Roberto Romano.

L'édition française

La couverture de l'édition française a été créée par Hanna Dyer et Melanie Wang.

Cette traduction a été relue par **Guillaume Compagnon**.

Les fontes du texte de la version française sont Le Monde Livre et TheSansMonoCon.

Achevé d'imprimer par Corlet, Imprimeur, S.A. - 14110 Condé-sur-Noireau
N° d'Imprimeur : 62056 - Dépôt légal : décembre 2002 - *Imprimé en France*

	Débutant	Expérimenté	Spécialiste
Se former			
Agir vite			
Aide-mémoire			

Programmation

O'REILLY®

Servlets Java - Guide du programmeur

Les *servlets* Java excellent dans tous les domaines de la création d'information : que ce soit en réponse à des requêtes émises par un client web, à partir de documents XML ou directement en relation avec des bases de données, elles tirent parti de tous les avantages de Java en matière de portabilité, de puissance (multi-threading, connectivité, sérialisation), d'efficacité et d'endurance, de sécurité des données, d'intégration et de modularité, de souplesse et d'extensibilité.

Cet ouvrage a été remis à jour pour la version 2.3 de l'API Servlets qui porte Java à un nouveau niveau de développement pour le Web. En effet, les servlets se trouvent désormais au coeur de la spécification de la plate-forme J2EE et sont encore plus performantes. Ce livre vous montrera comment écrire rapidement les *servlets* les plus simples ou les plus complexes, qui tourneront sur la plupart des serveurs HTTP, d'*Apache* à *WebSite* en passant par *Java Web Server*, *IIS* ou *FastTrack*. Les cinq technologies les plus répandues de création de contenu basées sur les servlets sont abordées : JSP, Tea, WebMacro, XMLC et ECS. Cette nouvelle édition traite aussi du WAP.

Parmi les nombreux thèmes abordés :

- Le cycle de vie d'une servlet.
- L'envoi d'information HTML et de contenu multimédia.
- Le suivi de session.
- La sécurité.
- La connexion à une base de données.
- La communication applet-servlet.
- La collaboration de servlets.
- Les servlets d'entreprise et J2EE.

Que vous soyez débutant ou confirmé en programmation Java, vous trouverez dans ce livre quantité d'exemples très détaillés qui vous décriront comment servir du contenu dynamique, exploiter JDBC, communiquer avec des *applets*, servir des pages en plusieurs langues, ou construire des chaînes de *servlets* très spécialisées.

Jason Hunter est consultant Java. Il travaille actuellement chez *CollabNet*. Il est aussi membre expert du groupe responsable du développement de Tomcat.

William Crawford conduit une équipe de développeurs chez *Invantage Inc.*, spécialisée dans les outils Intranet basés sur Java.

ISBN 2-84177-196-2

9 782841 771967

www.oreilly.fr